SUBMICRON SYSTEMS ARCHITECTURE

SEMIANNUAL TECHNICAL REPORT

Sponsored by
Defense Advanced Research Projects Agency
ARPA Order Number 3771

Monitored by the
Office of Naval Research
Contract Number N00014-79-C-0597

5178:TR:85

Computer Science
California Institute of Technology

March 1985

Submicron Systems Architecture

Semiannual Technical Report

Computer Science
California Institute of Technology

5178:TR:85

March 1985

Reporting Period:	16 October 1984 to 15 March 1985
Principal Investigator:	Charles L Seitz
Faculty Investigators:	James T Kajiya
	Alain J Martin
	Robert J McEliece
	Martin Rem
	Charles L Seitz
	Henk Van Tilborg

Sponsored by the
Defense Advanced Research Projects Agency
ARPA Order Number 3771

Monitored by the
Office of Naval Research
Contract Number N00014-79-0597

SUBMICRON SYSTEMS ARCHITECTURE

Computer Science
California Institute of Technology

1. Overview and Summary

1.1 Scope of this Report

This document reports the research activities and results for the seven month period 16 October 1984 to 15 March 1985 (5 months) under the Defense Advanced Research Project Agency (ARPA) Submicron Systems Architecture Project.

1.2 Objectives

The central theme of this research is the architecture and design of VLSI systems appropriate to a microcircuit technology scaled to submicron feature sizes, and includes related efforts in concurrent computation and VLSI design. Additional background information can be found in previous semiannual technical reports [5052:TR:82], [5078:TR:83], [5103:TR:83], [5122:TR:84], 5160:TR:84.

1.3 Highlights

Some highlights of the previous 7 months are:

- Cosmic cube technology transfer (2.1.1).
- First working Mosaic RAMs (2.2).
- Concurrent Prolog process model (3.1).
- Denotational Semantics for Prolog (3.1).
- New method for compiling programs into self-timed systems (3.6).
- New results in burst error codes (4.1).
- First benchmarks on the MOSSIM Simulation Engine (4.3).

2. Architectural Experiments

2.1 Cosmic Cube

W C Athas, Reese Faucette, Wen-King Su, Chuck Seitz

2.1.1 Technology Transfer

We reported in our last semi-annual technical report that a non-exclusive license had been negotiated for the cosmic cube, covering design and patent rights, and a resale license of the operating system.

We are now at liberty to report that this licensee is Intel Corporation. Intel announced the "iPSC" (Intel Personal Supercomputer) line of computers, versions connected as 5-cubes, 6-cubes, and 7-cubes (32, 64, and 128 nodes, respectively), on 11 February 1985, with first deliveries scheduled for June 1985. These systems are about 4 times as powerful per node than the cosmic cube. The instruction processor is an Intel 80286/80287 rather than the 8086/8087, the storage is 512K bytes rather than 128K bytes, and the communication channel rate is 10 Mbits/s instead of 2 Mbits/sec.

The technology transfer effort has been relatively straightforward. The Intel machines are expected to be very nearly (C) program compatible with the cosmic cube in spite of hardware differences, due to the resident operating system hiding hardware details successfully.

The terms of the license agreement include a contribution of an iPSC d7 (7-cube) machine that will be operated by the Caltech Submicron Systems Architecture Project. This machine will be available on the ARPAnet for programming and application experiments.

2.1.2 Retrospective and Future Plans

The cosmic cube project has reached a stage in which system-building has yielded to the fascination of programming and using this computing system. It is more an more being treated as a part of our regular computing environment. Thus it is time to offer some reflections on what we have learned from this architectural experiment, and to report where we plan to go from here.

We are quite pleased with the computational model and the corresponding primitives provided by the cosmic cube's operating system, the "cosmic kernel". We have come to consider them to be fundamental to ensemble machines of this grain size.

The queued communications with no acknowledgement of message receipt is a very weakly synchronized form of communication. When this property is strengthened with message order being preserved between any pair of processes, the message primitives are (1) very well suited to a broad class of typical programs, and (2) capable of expressing, if needed, the strongest forms of synchronization.

Another crucial property of the message primitives is that message sending and receiving is handled as a concurrent activity of a process. Hence, a process can manage the sending and receiving of many concurrent messages.

Other features of the message system are motivated by portability: (1) there is just one format for messages of any length, whatever the underlying protocols; (2) messages are sent and received whole; and (3) the semantics of the message operations are independent

of process placement. Thus the expression of a concurrent computation for the cosmic cube is quite general and is reasonably portable between different machines, concurrent or sequential, that support this same multiple process message passing environment.

Future development is expected to follow along two general lines.

We plan to migrate some of the lower level functions of cosmic kernel into a custom communication chip. From what we have learned in the design, implementation, and use of the cosmic cube message system, we now see way to manage the flow control and routing of messages that reduces the average communication latency and also lends itself very well to a VLSI. Work already has started on a chip called the "wormwhole chip" to handle message routing inside a cosmic cube, thus relieving the processor of this task. This chip uses the same e-cube routing algorithm that is used in the cosmic cube. Unlike the current system, which reads a message completely in before forwarding it, the wormwhole chip moves a message a flow-control packet at a time through a node. Thus the path of a message is analogous to a worm weaving its way across the machine. The worm's head can be at the destination while the tail is still entering the network. There is no queueing inside the routing chip aside from the single packet that is buffered in each channel of the chip, so all queueing must be accomplished at the source and destination. This technique has been simulated for much higher message rates than are typical of cosmic cube programs, and for cubes as large as 12 dimensions, in a program that runs on the 6-cube.

Second, we plan to use the primitives of Cosmic Kernel as building blocks for designing new programming languages for the cosmic cube. Our current efforts have been to provide a C programming environment, since as a system's programming language, C meshes well with the cosmic kernel primitives. A comprehensive C user's guide to the cosmic cube has been written, as well as many C programs. However, looking at message passing actions not as system calls, but as constructs of a programming language, raises many questions and suggests a number of opportunties. The cosmic cube's message operations, although included simply as C functions, actually create a concurrent process with shared variables. The descendant process persists until the specified message operation is completed.

Program flow analysis, such as live variable analysis, can be applied even to our present C programs. The result of program flow analysis would permit the automatic insertion of the code necessary to test message locks as well as the automatic setting up of message descriptors, thus relieving the programmer from a good deal of the tedium associated with such functions. Experience with this approach to the general problem of how to express concurrent computations within this process model is leading to a concurrent object-oriented programming model that may well justify the design and implementation of a programming language particularly for such machines.

2.1.3 Hardware and Software Status

Our 6-cube (64 node) and 3-cube (8 node) Cosmic Cubes continue to run reliably, the last hard failure having occurred 10 months ago. With at this time a total of over 1,000,000 node-hours of operation and only two hard failures logged, both in dRAM chips, the calculated MTBF of the nodes of 100,000 hours can be regarded as conservative at the 99.9% confidence level, and an MTBF of 200,000 hours can be stated at a 95% confidence.

A second of the "rosebud" type of Cube Life Support board was built so that the interface to the 3-cube and 6-cube are identical. The current 6-cube host is a SUN file server, ARPAnet address `cit-sol`, and the current 3-cube host is a SUN workstation, ARPAnet address `cit-neptune`.

Both machines continue to be heavily used, even though all the Caltech physics department use has, as planned, been phased out.

The cosmic cube's resident operating system, the "cosmic kernel", is complete, but continues to be refined, particularly in the areas of performance tuning, debugging and error handling features, and the host software. Documentation of the system for users is also complete, and also continues to be refined.

With the cosmic kernel and host tools now quite stable and reliable, we are planning to experiment with time- and space-sharing of the 6-cube. The present "cube daemon" that runs in the SUNs as a user process will be enhanced to run as a system program that can serve multiple users.

2.1.4 Programming Techniques

Several programs intended to exempify and demonstrate programming techniques have been written, including:

- A broadcast spawning process, which, in addition to illustrating programming techniques, has become a "library" process to speed up program loading. (Faucette)
- A simulation of the time evolution of many-body systems that interact by symmetrical or unsymmetrical forces. (Seitz & Su)
- Programs to compute solutions to LaPlace's equation by relaxation, a classic program for parallel machines. (Su)
- A numerical sieve for the "Collatz problem", which differs from previous such programs both in using concurrency and by examining classes of integer expressions rather than individual integers. (Seitz)
- A simulation of the message traffic for "wormhole" routing in a binary n-cube, and using a different message flow control and routing technique (see 2.1.2) than is used in the cosmic cube. (Su)

Some other programs being written by the ARPA group include:

- A concurrent solution of the n queens problem. (Dally)
- A graphics package. (Su)
- Game playing programs. (Several)

Some of the programs being written for reasons other than illustrations of programming technique are described in later sections.

In addition, we continue to make the cosmic cube available to selected guest researchers for problems that appear to be particularly interesting or difficult, including in the last reporting period: guests from the Caltech engineering division doing jet plume computations, and guests from NSA.

2.2 Mosaic Systems

Steve Rabin, Don Speck, Chuck Seitz

Work is proceeding on schedule, in spite of a setback in our first Mosaic RAM chips, for assembling a 1024-node system based on a single chip Mosaic node. The principal effort in the past 5 months has been in discovering the sources and remedies for problems observed in

the Mosaic RAM chips. These chips use a devious circuit design style known as "hot-clock *n*MOS", described in an attached paper.

The first prototype dynamic RAM module (later to be integrated with the Mosaic processor and communication channels) was received from MOSIS, tested, and found to have a geometrical design rule error that obscured the test results. The layout was fixed, and a second iteration was fabricated. The first chips from this second design iteration were able to read but not write. This problem was eventually traced to a clocked-isolation clock-AND circuit (see figure 9 in the attached paper) that were producing an output pulse when their *enable* input was 0.

Chips from the next MOSIS run, which has a fairly large t_{ox} of 65 nm and a very high enhancement threshold voltage of 1.2 volts, worked correctly, although not as fast as we had expected (7 MHz instead of 20 MHz clock rate). It is interesting to us that we obtained for this "out of spec" MOSIS run some design information that was vital to getting back on schedule with this project.

From this testing and from SPICE simulations, we have learned how to broaden the tolerance of these circuits to variations in process parameters over a range that should be completely safe for MOSIS processes.

The effort for the top assembly and packaging of the 1024-node Mosaic is underway.

3. Concurrent Computation

3.1 Concurrent Prolog Process Model

Pey-yun Peggy Li and Alain J Martin

Due to its expressive power and simple semantics, Prolog becomes a promising language for many applications. Moreover, logic programs are inherently well suited to parallel computations. A Concurrent Prolog Process Model is proposed to carry out OR parallelism, AND parallelism, and stream parallelism inherent in a Prolog program. The target machine of this Process Model is the Sneptree. Hence, distributed computation without shared memory and with message passing synchronization is assumed.

The Process Model contains two types of processes, AND processes and OR processes, which are alternatively connected into a proof tree. There exist communication channels between sibling AND processes to transmit the values of shared variables. The AND parallelism is realized by a data driven model and an input buffer merge mechanism. One major distinction of our model from previous models is that the processes in our model generate all the solutions at once, send them out to where they are requested, and terminate themselves immediately. A smart merge algorithm in the receiving process then combines and synchronizes the partial solutions from different places to form correct results. With this mechanism, we can avoid complicated backward execution in previous models, and thus more parallelism, less overhead, and simpler control can be achieved.

An extension of standard Prolog that integrates the concepts of Shapiro's Concurrent Prolog and Wise's Epilog is used as the base language in the Concurrent Process Model. An ordering algorithm is designed to construct the data flow diagram of the AND processes and detect the possible multiple paths between any two AND processes. Hence, the message channels between two AND processes or one AND processes and its father can be built up. Necessary synchronization signals are also generated by proper processes. A merge algorithm is also designed to either gather all the partial solutions from its descendants in an OR process or combine the input variable bindings from its sibling AND processes in an AND process. The merge algorithm does a Cartesian Product over all the input streams. With the appearance of synchronization signals, it does several Cartesian Products against the proper synchronization signals.

Mapping of our Process Model onto a Sneptree is going to be investigated next. Since a Sneptree can simulate an arbitrary size binary tree optimally, we expect that our model can be mapped onto the Sneptree with very little overhead.

3.2 Inverse Limit Construction of Infinite Lists

Young-il Choo, Alain J Martin

Infinite lists arise when we want to give meaning to non-terminating, yet answer producing, programs. For applicative languages this is possible using the lazy evaluation mechanism. Precise formulation of infinite lists is necessary for formal reasoning of such programs.

Infinite objects can be computed only as a limit of finite ones. The notion of a limit presupposes some kind of a topology, or at least an ordering. The ordering we use is that of definedness. By generating a sequence of finite lists that become more and more defined,

we can specify an infinite list. This notion is embodied in a construction from topology called the inverse limit.

We have constructed a domain of infinite lists by taking the inverse limit of the chain of domains of finite lists ordered by projection. The domains contain finite lists up to a certain length, and the projection is that of the best approximation in the domain below. We have proved that the resulting domain is a complete partial order, and therefore, can be used as a semantic domain for the fixed point approach to program semantics.

These results will appear in a report "Notes on infinite lists I: Inverse limit construction".

Future work will explore the topological structure of the domain of infinite lists, and consider the connections with Scott's domain for λ-calculus which is also constructed using the inverse limit.

3.3 Denotational Semantics of Prolog

Young-il Choo, Jim Kajiya

The denotation of a Prolog program is a function (called substitution) that maps variables to terms of the Herbrand universe. The standard ordering defined componentwise on the variables makes incomparable those which differ at most by renaming of variables.

We have defined the ordering between substitutions that uses the category theory notion of one function factoring through another, i.e. $f \sqsubseteq g$ iff there exists h such that $g = h \circ f$. This enables a clean fixed point approach to Prolog type languages where the nondeterminism is taken care of by the function compositions.

Once the semantic domain is defined, we can verify Prolog programs by algebraic transformations.

3.4 Process Placement

Craig Steele, Chuck Seitz

This effort is directed at optimization of process placement on distributed homogeneous processor architectures, and related comparative studies of different direct networks.

A process model computation may be represented as a graph, where communicating processes are the vertices, and logical communication channels are the edges. Likewise the underlying physical structure may be represented as a graph with processors and physical communication paths represented as vertices and edges. To run a computation on a distributed processor, the processes must be loaded onto the physical machine, requiring a mapping of the logical graph onto the physical graph.

Minimizing the communication cost of the computation is a major problem unless the logical and physical graphs are similar. While many problems of interest can take advantage of the isomorphism of the n-cube to three-space, even in three-dimensional simulations the density of simulated objects may vary, lowering efficiency with simple partitioning. Graph structures common in applications of interest in computer science, such as trees, do not map in obviously good ways to n-cube architectures.

Using the technique of simulated annealing, good mappings may be found in moderate time for arbitrary logical computation graphs. Taken into account are arbitrary edge weights

(reflecting varied utilization of logical communication channels) and arbitrary process memory sizes (allowing the physical processor constraints to affect the density of processes per processor).

This method has been applied to a comparative study of proposed interconnection structures for homogeneous machines for both single and multiple process to processor mappings. Efficient mappings are found for interconnects less costly than binary n-cubes for most problems at only modestly increased communication costs. For example, a modified shuffle-exchange interconnect for the 1024-element Mosaic design has been simulated for equally large problems to good effect.

Work is continuing to extend the architectural comparison to include congestion factors for networks in which communication channel saturation is significant.

Future work will include modification of the existing program to allow the optimized mappings it generates to direct loading of processes into the Cube. User-defined labels for vertices and edges will allow a more friendly style of specifying process connections.

Experiments to determine the convergence rates of the program using more localized cost functions and with informational 'noise' will determine the suitability of the method for concurrent implementation.

3.5 Concurrent Data Structures

Bill Dally, Chuck Seitz

The appearance of concurrent computers such as the Cosmic Cube and the Mosaic is creating a need for concurrent data structures and algorithms. While a great deal of work has been done on concurrent algorithms for numerical problems, very little is known about concurrency in non-numerical applications. Conventional data structures such as heaps and B-trees have many bottlenecks which limit their potential concurrency and make them unable to take advantage of the computing potential of these concurrent machines. New data structures are required to harness the power of concurrent computing.

Our research has been proceeding in two directions: to study the concurrency limitations of existing data structures and to develop new data structures which overcome these limitations. A heap was selected as representative of existing data structures and its concurrency properties were studied in detail [5156:DF:84]. As a result of this research some variants of conventional heaps were developed which offer improved concurrency and virtual memory performance. However, even these variant heaps are limited by their tree structure to concurrency which grows only as the log of the number of elements in the heap.

To overcome the concurrency limitations of conventional data structures we have developed two new data structures for ordered sets: the balanced cube and the B-cube [5174:TR:85]. By using a binary n-cube rather than a tree to organize data, these structures overcome the bottleneck of a single entry point or root node. As a result, the concurrency of these cube structures was thought to grow almost linearly with the number of elements in the set. A set of experiments has been run to test the balanced cube algorithms. The tests showed that the concurrency of the cubes grows $O(N/logN)$, not linearly as originally predicted. This sublinear performance is due to a mismatch between the topology of the cube and the order relation in the set. An obvious way to correct the mismatch is to add a ring connection to the binary n-cube computer; however the adjacency perserving mapping described below is a more attractive solution.

A novel method of mapping the elements of an ordered set to the processing nodes of a concurrent computer using a Gray code has been developed. [5176:DF:85]. The Gray code preserves adjacency. If the Gray code mapping is used, two elements which are neighbors in linear space are mapped to neighboring processors on the cube. Because of the reflection properties of the Gray code, a balanced cube using this mapping is also easier to search than a direct mapped cube.

Current work on concurrent data structures is concentrating on graph algorithms. Concurrent algorithms have been developed for the shortest path problem and for graph searching problems. Experiments to test these algorithms are in progress.

3.6 Self-timed Design

Alain J Martin

A paper describing a method for compiling a set of communicating processes into a self-timed circuit, including an illustration of its application to a distributed mutual exclusion algorithm, is attached.

3.7 Wormhole routing

Wen-King Su, Chuck Seitz

A new technique for flow control and routing in a binary n-cube was devised. In this technique no more than a flow control packet, typically 32 bits, is queued in a node of the n-cube. Messages flow directly through a node unless the required outgoing channel is already occupied, in which case the blocking of channels propagates back to the source of the message. In combination with the e-cube routing algorithm this technique can be shown to be free of deadlock.

However, because of the propagation of blocking, it is not intuitive that the performance of this routing scheme would be satisfactory, nor is the message flow analytically tractable. Some early simulations on an IBM PC suggested, however, that the performance of this routing scheme is excellent. A full scale simulation program was written for the cosmic cube, and run on the 6-cube for connections up to 12-cubes (4096 nodes). A more complete report will follow a series of simulation runs to gather complete statistics.

3.8 Multiprocessor Architectures for High Performance Computer Graphics

Daniel S Whelan, Jim Kajiya

My work during the last quarter has concentrated on the design and implementation of multiprocessor algorithms for producing images of synthetic scenes with shadowing effects. With the exception of some ray tracing architectures, no multiprocessor architectures proposed for computer graphics support shadowing effects. While ray tracing architectures produce extremely realistic images, they do so rather slowly making them inadequate for real-time simulation applications.

Shadowing is inherently a non-local effect, thus my work has concentrated on partitioning the shadowing computations such that interprocessor communications is both localized and minimized. My approach separates shadowing computations into two distinct processes which I call local shadowing and foreign shadowing.

Local shadowing determines whether any object within a processor's view space casts a shadow upon any other visible local object. Foreign shadowing determines whether objects outside the processor's view space shadow objects within the processor's view space.

My implementation of a foreign shadowing algorithm partitions well onto a multiprocessor with only very local interprocessor communications. The performance of this shadowing algorithm should far exceed the performance of ray tracing algorithms. The overall architecture will provide performance well in excess of that of today's flight simulation engines and will provide features such as shadows which today's flight simulators do not implement properly.

4. VLSI Design

4.1 Soft Error Correction for Increased Densities in VLSI Memories

Robert J McEliece, Khaled Abdel-Ghaffar, Henk Van Tilborg

In previous work, we have studued what might be called the "ultimate" physical limits of densities in VLSI RAMS, studying in detail the restrictions imposed by quantum and thermodynamic effects, for example. Recently, however, we have turned the more immediately worrisome effects of alpha-particles and other types of "soft" errors.

At sub-micron feature sizes, a single soft error event will in general cause not just a single bit error, but a two-dimensional "patch" or "burst" of errors, perhaps on the order of 2 - 4 bits square. Thus we have been focusing or most recent research on the problem of two-dimensional burst-error correcting codes, a subject which has until our own work not received much attention in the literature.

We have devised three new classes of two-dimensional burst correction codes, each of which is more efficient than any previously studied class. The first class is based on the new idea of "burst identification." A code in this first class is constructed from two simple sub-codes, one which is able to identify the burst pattern, and one which is able to locate it once it is identified.

The second class is based on extending the well-known Fire codes, which are successfully used to correct ordinary one-dimensional bursts. While the notion of "two-dimensional Fire codes" has appeared in the literature, our generalization appears to be superior to the older one, both from the standpoint of minimizing overhead and ease of implementation.

The third class is an ad hoc class of good two-dimensional cyclic codes found by computer-aided search. For the small burst sizes likely to be important in VLSI applications, this class actually appears to be the most promising.

Finally, we mention that as an unexpected by-product of our research, we have recently been able to prove a long-standing conjecture about one- dimensional cyclic burst error-correcting codes, viz. that for fixed burst length, the Hamming bound gives asymptotically the minimum achievable redundancy. (This work has been done jointly with Andrew Odlyzko of AT&T Bell Labs.)

We are preparing several papers for external publication to report our findings. A preliminary version of the paper "two-dimensional burst identification codes" was presented by Robert McEliece at the 1985 IEEE workshop on Information theory, held at Caesaria, Israel, last summer; and a preliminary version of the work on one- and two- dimensional cyclic burst error correction will be given by McEliece in April 1985 at the 10th Carribean Conference on Combinatorics and Computers.

4.2 CAD Tools

Steve Rabin

Miscellaneous problems with the IC verification tools have been uncovered and fixed. The artwork circuit extracter has been upgraded to run under version 8 Mainsail, ported to several our new machines, and optimized to run 30% faster. Our design tools are now capable of verifying bulk CMOS circuit designs.

4.3 Mossim Simulation Engine

Bill Dally

Design verification is essential in the development of VLSI systems. The complexity of VLSI circuits, inaccessability of internal nodes, and difficulty of repair make the probability of producing a working chip very low without extensive design verification. As the complexity of VLSI circuits approaches 10^6 devices, the computational requirements of design verification are exceeding the capacity of general purpose computers. To provide the computing power required to verify these complex VLSI chips, we are developing the Mossim Simulation Engine (MSE) [5123:TR:84].

The Mossim Simulation Engine is a special purpose processor which, in a single processor configuration, performs switch-level simulation of MOS VLSI circuits 200-500 times faster than a VAX 11/780. In multiple processor configurations even greater speedup can be achieved.

The MSE overcomes two limitations of existing simulation engines. By using the switch-level model developed by Bryant [5065:TR:83], the MSE performs accurate simulation of MOS circuits. Existing simulation engines perform logic simulation and cannot model MOS effects such as stored charge, charge sharing and transistor ratios. Also, by using the concept of virtual processors the MSE can simulate a circuit many times larger than the size of the processor. Existing simulation engines are limited to simulating circuits which fit in the processor.

A prototype MSE has been constructed and is now working. Initial experiments on the prototype machine show a measured performance of 150K gate evaluations per second only slightly less than the performance estimated by simulation. The only MSE software currently available is a hardware debug monitor. Systems software: a run time system and a user interface, to make the MSE available to VLSI designers is being planned.

The prototype will be used both as a research tool to support the simulation requirements of the next generation of VLSI circuits and as a test bed for experiments in switch-level simulation including: the development of a virtual simulation processor system, experiments in the application of multi-processing to switch-level simulation, and a study of the locality of activity in MOS circuits.

A paper describing the MSE is attached.

California Institute of Technology
Computer Science, 256-80
Pasadena, California 91125

ARPA Technical Memos and Technical Reports
together with selected other Caltech reports on VLSI topics
March 1985

Available from the Computer Science Department Library
+++

5178:TR:85 "Submicron Systems Architecture," March 1985; ARPA Semiannual Technical Report

5177:TR:85 "Hot-Clock nMOS," March 1985; Seitz, Charles L, et al

5174:TR:85 "The Balanced Cube: A Concurrent Data Structure," February 1985; Dally, William J and Charles L Seitz

5172:TR:85 "A Combined Logical and Functional Programming Language," January 1985; Newton, Michael O.

5168:TR:85 "An Object Oriented Architecture," December 1984; Dally, William J and James T Kajiya

5143:TR:84 "The General Interconnect Problem," MS Thesis, May 1984; Ngai, John [$5.00]

5139:TR:84 "HEX: A Hierarchical Circuit Extractor," MS Thesis, May 1984; Oyang, Yen-Jen [$4.00]

5137:TR:84 "The Dialogue Designing Dialogue System," PhD Thesis, May 1984; Ho, Tai-Ping [$7.00]

5136:TR:84 "Heterogeneous Data Base Access," PhD Thesis, Papachristidis, Alex [$5.00]

5135:TR:84 "Toward Concurrent Arithmetic," MS Thesis, April 1984; Chiang, Chao-Lin [$7.00]

5134:TR:84 "Using Logic Programming for Compiling APL," MS Thesis, April 1984; Derby, Howard [$2.00]

5132:TR:84 "Switch Level Faults," April 1984; Schuster, Mike [$10.00]

5130:TR:84 "LOG The Chipmunk Logic Simulator User's Guide," April 1984; Gillespie, Dave [$3.00]

5129:TR:84 "Design of the MOSAIC Element," MS Thesis, April 1984; Lutz, Chris [$5.00]

5125:TR:84 "Supermesh," MS Thesis, March 1984; Su, Wen-king [$4.00]

5123:TR:84 "The Mossim Simulation Engine Architecture and Design," March 1984; Dally, Bill [$5.00]

5122:TR:84 "Submicron Systems Architecture," March 1984; ARPA Semiannual Technical Report [$3.00]

5120:TM:84 "A Mathematical Approach to Modeling the Flow," March 1984; Johnsson, Lennart and Danny Cohen [$1.00]

5118:TR:84 "SMART User's Guide," February 1984; Ngai, John [$2.00]

5116:TM:84 "Design Automation and Engineering Organization Structures," February 1984; Segal, Richard [$1.00]

5113:TR:84 "The Wo$_L$ery," January 1984; Mead, Carver A. [$2.00]

5112:TR:83 "Parallel Machines for Computer Graphics," PhD Thesis, January 1983; Ulner, Michael [$22.00]

5105:TR:83 "Memory Management in the Programming Language ICL," November 1983; Wawrzynek, John [$2.00]

5104:TR:83 "Graph Model and the Embedding of MOS Circuits," MS Thesis, Nov. 1983; Ng, Tak-Kwong [$6.00]

5103:TR:83 "Submicron Systems Architecture," Nov. 1983; ARPA Semiannual Technical Report [$3.00]

5102:TR:83 "Experiments with VLSI Ensemble Machines," October 1983; Seitz, Charles L [$2.00]

5101:TM:83 "Concurrent Fault Simulation of MOS Digital Circuits," October 1983; Bryant, Randal E [$1.00]

5099:TM:83 "VLSI and the Foundations of Computation," September 1983; Mead, Carver [$1.00]

5098:TM:83 "New Techniques for Ray Tracing Procedurally Defined Objects," September 1983; Kajiya, James T [$2.00]

5094:TR:83 "Stochastic Estimation of Channel Routing Track Demand," July 1983; Ngai, John [$2.00]

5093:TR:83 "Design of the MOSAIC Element," July 1983; Lutz, Chris, Steve Rabin, Chuck Seitz and Don Speck [$1.00]

5092:TM:83 "Residue Arithmetic & VLSI," July 1983; Chiang, Chao-Lin and Lennart Johnsson [$2.00]

5091:TR:83 "Race Detection in MOS Circuits by Ternary Simulation," June 1983; Bryant, Randal E [$2.00]

5090:TR:83 "Space-Time Algorithms: Semantics and Methodology," PhD Thesis, June 1983; Chen, Marina Chien-mei [5.00]

5089:TR:83 "Signal Delay in General RC Networks with Application to Timing Simulation of Digital Integrated Circuits," July 1983; Lin, Tzu-Mu and Carver A Mead [$6.00†

5086:TR:83 "A VLSI Combinator Reduction Engine," MS Thesis, May 1983; Athas, William C Jr [$4.00]

5084:TM:83 "The Tree Machine: An Evaluation of Strategies for Reducing Program Loading Time," Aug. 1983; Li, Pey-yun Peggy, and Lennart Johnsson [$3.00]

5082:TR:83 "Hardware Support for Advanced Data Management Systems," PhD Thesis, April 1983; Neches, Philip [$10.00]

5081:TR:83 "RTsim - A Register Transfer Simulator," April 1983; Lam, Jimmy [$4.00]

5079:TR:83 "Highly Concurrent Algorithms for Solving Linear Systems of Equations," April 1983; Johnsson, Lennart [$2.00]

5078:TR:83 "Submicron Systems Architecture," April 1983; ARPA Semiannual Technical Report [$5.00]

5075:TR:83 "A General Proof Rule for Procedures in Predicate Transformer Semantics," March 1983; Martin, Alain [$2.00]

5074:TR:83 "Robust Sentence Analysis and Habitability," March 1983; Trawick, David [$10.00]

5073:TR:83 "Automated Performance Optimization of Custom Integrated Circuits," February 1983; Trimberger, Steve [$10.00]

5068:TM:83 "A Hierarchical Simulator Based on Formal Semantics," Proc Third Caltech Conf on VLSI, March 1983; Chen, Marina and Carver Mead [$1.00]

5065:TR:82 "Switch Level Model & Simulator for MOS Digital Systems," July 1983; Bryant, Randal E. [$3.00]

5059:TM:82 "A Comparison of MOS PLAs," December 1982; Trimberger, Stephen [$2.00]

5055:TR:82 "FIFO Buffering Transceiver: A Communication Chip Set for Multiprocessor Systems," MS Thesis, December 1982; Ng, Charles H [$5.00]

5052:TR:82 "Submicron Systems Architecture," October 1982; ARPA Semiannual Technical Report [$4.00]

5049:TR:82 "Distributed Mutual Exclusion Algorithms," submitted for publication, AJM 31, September, 1982, Martin, Alain [$3.00]

5047:TR:82 "The Torus: An Exercise in Constructing a Processing Surface," Proc 2nd Caltech Conference on VLSI, January 1981; Martin, Alain [$3.00]

5046:TR:82 "An Axiomatic Definition of Synchronization Primitives," Acta Informatica 16, pp 219-235 (1981), Martin, Alain [$3.00]

5045:TM:82 "A Distributed Implementation Method for Parallel Programming," Proc Information Processing '80, Martin, Alain [$3.00]

5044:TR:82 "Hierarchical Nets: A Structured Petri Net Approach to Concurrency," September 1982; Choo, Young-Il [$10.00]

5043:TM:82 "A Formal Derivation of Array Implementations of FFT Algorithms," Proc USC Workshop on VLSI & Modern Signal Processing, November 1982; Johnsson, Lennart and Danny Cohen [$3.00]

5042:TR:82 "Formal Specification of Concurrent Systems: VLSI, Signal Processing, and Formal Semantics; Concurrent Algorithms as Space Time Recursion Equations," Proc USC Workshop on VLSI & Modern Signal Processing, November 1982; Chen, Marina and Carver Mead [$4.00]

5040:TR:82 "Concurrent Algorithms for the Conjugate Gradient Method," September 1982; Johnsson, Lennart [$3.00]

5038:TM:82 "A New Channel Routing Algorithm," September 1982; Chan, Wan S [$4.00]

5035:TR:82 "Type Inference in a Declarationless, Object-Oriented Language," MS Thesis, June 1982; Holstege, Eric [$10.00]

5034:TR:82 "Hybrid Processing," PhD Thesis, March 1982; Carroll, Chris [$12.00]

5033:TR:82 "MOSSIM II: A Switch-Level Simulator for MOS LSI User's Manual," August 1982; Schuster, Mike, Randal Bryant and Doug Whiting [$4.00]

5030:TM:82 "VLSI Algorithms for Doolittle's, Crout's, and Cholesky's Methods," Proc ICCC '82 IEEE Int'l Conf on Circuits & Computers, September 1982; Johnsson, Lennart [$1.00]

5029:TM:82 "POOH User's Manual," July 1982; Whitney, Telle [$4.00]

5027:TM:82 "Concurrent Programming," July 1982; Bryant, Randal E and Jack B Dennis [$3.00]

5021:TR:82 "Earl: An Integrated Circuit Design Language," MS Thesis, May 1982; Kingsley, Chris [$5.00]

5019:TR:82 "A Computational Array for the QR-Method," Proc MIT Conf on Advanced Research in VLSI, January 1982; Johnsson, Lennart [$3.00]

5018:TM:82 "Filtering High Quality Text for Display on Raster Scan Devices," August 1981; Kajiya, Jim and Mike Ullner [$2.00]

5017:TM:82 "Ray Tracing Parametric Patches," May 1982; Kajiya, Jim [$2.00]

5016:TR:82 "Bristle Blocks - Scrutinized and Analyzed," June 1982; McNair, Richard and Monroe Miller [$4.00]

5015:TR:82 "VLSI Computational Structures Applied to Fingerprint Image Analysis," Megdal, Barry [$15.00]

5014:TR:82 "The Extension of Object-Oriented Languages to a Homogeneous, Concurrent Architecture," PhD Thesis, April 1982; Lang, Charles R Jr [$15.00]

5012:TM:82 "Switch-Level Modeling of MOS Digital Circuits," Bryant, Randal [$2.00]

5005:TM:82 "Chip Assembly Tools," Trimberger, Steve and Chris Kinglsey [$2.00]

5004:TM:82 "Riot - A Simple Graphical Chip Assembly Tool," Trimberger, S. and Jim Rowson [$2.00]

5003:TM:82 "Pipelined Linear Equation Solvers & VLSI ," Proc Microelectronics 1982, May 1982; Johnsson, Lennart S. [$2.00]

5001:TR:82 "Minimum Propagation Delays in VLSI ," IEEE J Solid State Circuits, August 1982; Mead, Carver, and Martin Rem [$2.00]

5000:TR:82 "A Self-Timed Chip Set for Multiprocessor Communication," MS Thesis, February 1982; Whiting, Douglas [$6.00]

4778:TM:82 "Testing and Structured Design," Proc Int'l Test Conf, August 1982; DeBenedictis, Erik P and Charles L Seitz [$2.00]

4777:TR:82 "Techniques for Testing Integrated Circuits," PhD Thesis, DeBenedictis, Erik P [$7.00]

4716:TM:82 "A Rectangular Area Filling Display System Architecture," Whelan, Dan [$4.00]

4684:TR:82 "A Characterization of Deadlock Free Resource Contentions," Chen, Marina, Martin Rem, and Ronald Graham [$3.00]

4675:TR:81 "Switching Dynamics," MS Thesis, Lewis, Robert K [$7.00]

4655:TR:81 "Proc Second Caltech Conf on VLSI," January 1981; Seitz, Charles, ed. [$20.00]

4654:TR:81 "A Versatile Ethernet Interface," MS Thesis, Whelan, Dan [$12.00]

4653:TR:81 "Toward A Theorem Proving Architecture," MS Thesis, Lien, Sheue-Ling [$10.00]

4618:TM:81 "The Tree Machine Operating System," Li, Peggy [$5.00]

4600:TM:81 "A Notation for Designing Restoring Logic Circuitry," Proc Second Caltech Conf on VLSI, January 1981; Rem, Martin, and Carver Mead [$3.00]

4530:TR:81 "Silicon Compilation," PhD Thesis, Johannsen, Dave [$18.00]

4527:TR:81 "Communicative Databases," PhD Thesis, Yu, Kwang-I [$11.00]

4521:TR:81 "Lambda Logic," MS Thesis, Rudin, Leonid [$8.00]

4517:TR:81 "The Serial Log Machine," MS Thesis, Li, Peggy [$7.00]

4407:TM:82 "An Experimental Composition Tool," Mosteller, Richard C [$3.00]

4336:TR:81 "A Structured Design Methodology and Associated Software Tools," Trimberger, S with J Rowson, D Lang, and J P Gray [$3.00]

4334:TR:81 "An Inexpensive Multibus Color Frame Buffer," Whelan, Dan and R Eskanazi [$1.00]

4332:TR:81 "RLAP, Version 1.0, A Chip Assembly Tool," Mosteller, R [$3.00]

4320:TR:81 "A Hierarchical Design Rule Checker," MS Thesis, Whitney, Telle [$7.00]

4317:TR:81 "REST - A Leaf Cell Design System," MS Thesis, Mosteller, Richard C [$10.00]

4298:TR:81 "From Geometry to Logic," MS Thesis, Lin, Tzu-mu [$5.00]

4287:TR:81 "Computational Arrays for Band Matrix Equations," Johnsson, Lennart [$2.00]

4281:TR:81 "Combining Graphics and a Layout Language in a Single Interactive System," Trimberger, Steve [$2.00]

4204:TR:78 "A 16-Bit LSI Digital Multiplier," EE Thesis, Masumoto, R T [$8.00]

4191:TR:81 "Towards A Formal Treatment of VLSI Arrays," Proc Second Caltech Conf on VLSI, January 1981; Johnsson, Lennart S, Uri Weiser, D Cohen, and Alan L Davis [$4.00]

4168:TR:81 "Computational Arrays for the Discrete Fourier Transform," Proc 22nd Computer Science Int'l Conf CompCon 81, February 1981; Johnsson, Lennart S and D Cohen [$3.00]

4116:TR:79 "Toward A Mathematical Theory of Perception," PhD Thesis, Kajiya, Jim [$18.00]

4088:TR:80 "The Representation of Communication and Concurrency," Milne, George [$8.00]

4087:TR:80 "Gaussian Elimination of Sparse Matrices and Concurrency - A Complexity Analysis," Johnsson, Lennart [$3.00]

4029:TR:80 "Structure, Placement and Modeling," MS Thesis, Segal, Richard [$8.00]

4025:TM:80 "Sticks Standard Software Package," Segal, Richard and Steve Trimberger [$2.00]

4024:TM:80 "SSP Basic Software Package," Silicon Structures Project [$5.00]

4022:TM:80 "Comprehensive CIF Test Set," Trimberger, Steve [$2.00]

3901:TM:78 "Hierarchical Design for VLSI," Rowson, Jim [$1.00]

3882:TM:80 "A Chip Assembler," Tarolli, Gary [$2.00]

3880:TM:80 "The Proposed Sticks Standard," Trimberger, Steve [$5.00]

3857:TM:80 "VLSI Architecture & Design," Proc Nat'l Electronics Conf, October 1980; Johnsson, Lennart [$1.00]

3805:TR:80 "SSP Annual Report," Silicon Structures Project [$3.00]

3762:TR:80 "A Software Design System," PhD Thesis, Hess, Gideon [$8.00]

3761:TR:80 "A Fault Tolerant Integrated Circuit Memory," PhD Thesis, Barton, Tony [$7.00]

3760:TR:80 "The Tree Machine: A Highly Concurrent Computing Environment," PhD Thesis, Browning, Sally [$10.00]

3759:TR:80 "The Homogeneous Machine," PhD Thesis, Locanthi, Bart [$7.00]

3710:TR:80 "Understanding Hierarchical Design," PhD Thesis, Rowson, James [$9.00]

3642:TM:80 "Modeling and Verification in Structured Integrated Circuit Design," PhD Thesis, Buchanan, Irene [$10.00]

3487:TM:80 "The Proposed Sticks Standard," Trimberger, Steve [$2.00]

3364:TM:79 "Stack Data Engine," December 1979; Efland, G and R C Mosteller [$5.00]

3357:TM:79 "CIF20P Instruction Manual," Tarolli, Gary and Dick Lang [$3.00]

3356:TR:79 "LAP User's Manual," Lang, Dick [$3.00]

3353:TM:79 "FORTRAN Debugging Aids," Trimberger, Steve [$3.00]

3352:TM:80 "A Comprehensive CIF Test Set," December 1979; Trimberger, Steve [$2.00]

2883:TR:79 "A Pascal Machine Architecture Implanted in Bristle Blocks, a Prototype Silicon Compiler," MS Thesis, Seiler, Larry [$10.00]

2870:TM:79 "A Wire Oriented Mask Geometry Editor," Trimberger, Steve [$5.00]

2686:TR:80 "The Caltech Intermediate Form for LSI Layout Description," Sproull, Robert and Richard Lyon, revised by S.Trimberger [$2.00]

2276:TM:78 "A Language Processor and a Sample Language," Ayres, Ron [$12.00]

Hot-Clock *n*MOS

Charles L Seitz, Alexander H Frey[1], Sven Mattisson[2]
Steve D Rabin, Don A Speck, Jan L A van de Snepscheut[3]

Department of Computer Science
California Institute of Technology
Pasadena, CA 91125

1. Also affiliated with IBM Los Angeles Scientific Center.

2. At Caltech 1982-84; now at the Applied Electronics Department, Lund Institute of Technology, Sweden.

3. At Caltech 1983-84; now at the Department of Mathematics and Computing Science, Groningen University, The Netherlands.

Abstract: "Hot-Clock *n*MOS" is a style of design that has advantages in circuit energetics and performance. When the application of this style is carried to its limits, an *n*MOS chip is powered entirely from its clock signals. There are savings in area, delay, and power, even when the bootstrap circuits of this style are used together with conventional circuitry. We have used this technique in numerous small projects and test structures, and in 3 substantial projects fabricated through MOSIS.

1. Energetics

How is the power required by and dissipated on a MOS chip used? Even in CMOS technology, in which the static power is negligible, "dynamic" power is required to charge and discharge capacitances:

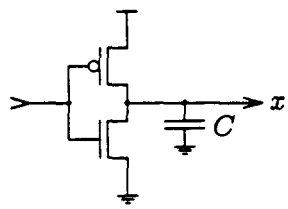


Figure 1

The research described in this paper was sponsored in part by the Defense Advanced Research Projects Agency, ARPA Order number 3771, monitored by the Office of Naval Research under contract number N00014-79-C-0597; and in part by IBM Corporation.

Each time the signal x is to change $0 \rightarrow 1$, the power supply must provide a quantity of charge CV_{dd} at potential V_{dd}, hence energy CV_{dd}^2. Half of this energy ends up stored in the capacitance C, and the other half is dissipated in the p-channel transistor. When x is to change $1 \rightarrow 0$, the charge stored on C is conducted through the n-channel transistor into the ground terminal, and the stored energy is dissipated in the transistor. Thus in a full cycle $0 \rightarrow 1 \rightarrow 0$ of signal x, energy CV_{dd}^2 must be supplied to the chip, and is dissipated in the transistors that drive this signal. The fundamental motivation behind hot-clock nMOS is to get around this "inevitable" dissipation of power on a high-complexity chip.

If one were to try to spot the places on a MOS chip where most of the dynamic power goes, it would be in the drivers of relatively large capacitances – long and/or highly loaded wires – that are driven at relatively high frequencies. Examples of such signals are control lines and data buses in instruction and arithmetic processors; word and bit lines in RAMs; literal and implicant lines in large PLAs, and output pads. By virtue of their capacitance, these are also signals that are difficult to drive with small delay.

Many of these signals are naturally driven in synchrony with one of the clock signals, so another possibility is to drive them through some approximation to an ideal switch:

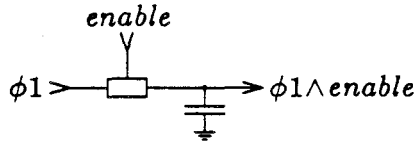


Figure 2

Here we assume that the output is initially 0, and that the *enable* signal changes only during $\phi 2$ in a two-phase non-overlapping clocking scheme. The output is then $\phi 1 \wedge$ *enable*. If the switch were ideal, the circuit would introduce no delay, the output being just a gated replica of the input clock. Also, the switch turns on only when there is no voltage across it, and off when there is no current flowing through it, so even if it did exhibit some non-zero resistance or conductance while switching, it dissipates no power in changing state.

Assume that when this switch is implemented with MOS transistors, it can be modeled as an ideal switch in series with an effective resistance

R, and that the clock transition can be modeled as a ramp from 0V to V_c in time t_r, and from V_c to 0V in time t_f:

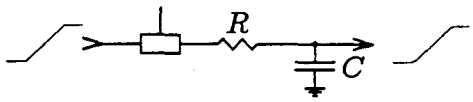


Figure 3

Let the switch be on. If t_r were 0, the full clock voltage would appear initially across the resistance, and the output node would be charged conventionally with the CV_c^2 energy from the clock being split between the output node capacitance and dissipation in the resistance. The case of interest to us is where t_r is some normal and achievable value – say a few ns –, and the sizes of the transistors that form the switch are selected so that R is small enough that the delay through the circuit, approximately RC, is much less than t_r. In this case in which $RC \ll t_r$, the switch is also sized to dissipate little power. The energy dissipated per switching event is closely approximated by:

$$E \approx \frac{RC}{t_r} CV_c^2.$$

Presume that the value of RC is fixed in the design, that is, that the sizes of the switch transistors are determined according to a given $RC \ll t_r, t_f$. Presume also that t_r and t_f are a fixed fraction of the clock period T. The same chip may then be operated at a longer clock period T, resulting in slower operation but also in a smaller *proportion* of the energy supplied in each clock transition being dissipated on-chip. In other words, these circuits exhibit a characteristic in which the total energy required to perform a computation varies as $1/T$. The computation is less costly if one is not in a hurry. This characteristic is at odds with complexity arguments that assert that the cost be expressed in E_{sw} units (§9.10 in [5]), and is interestingly similar to the AT^2 invariance exhibited by many algorithms [8], in which the cost AT of performing a given computation also varies as $1/T$.

This scheme does not really "solve" the power and speed problems of driving the capacitance C. Rather, we have *exported* the problem* elsewhere, namely, to whatever circuit drives the signal $\phi 1$.

* One might compare this technique with a country banning certain polluting industries, but still purchasing the products of those industries from other countries. The country may then be accused of "exporting pollution".

In hot-clock nMOS we export the problem off the chip. The off-chip clock driver can then use a technology that is better suited to driving the capacitive clock load than is the high complexity MOS technology. For example, bipolar transistors have much higher transconductance than MOS devices at present feature size, and make excellent clock drivers. A more interesting possibility is to use clock driver circuits that employ inductances. A resonant driver allows an almost lossless transfer of charge from the power supply to the clock capacitance, and then from the clock capacitance back into the power supply. Figure 4 presents an idealized circuit to implement this scheme of saving power in driving capacitive loads:

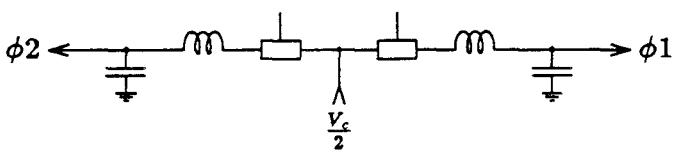


Figure 4

A practical circuit is rather more complicated. This trick is somewhat like "LC logic" (§9.1.3 in [5]), with the L's brought off the chip, and is subject to the same inevitability of some loss in the switching process. However, such techniques could allow VLSI systems – even in nMOS technology – to operate at dramatically lower overall dissipation levels than present CMOS circuits, let alone nMOS circuits.

2. The Elementary nMOS Clock-AND

This technique of exporting the problem of driving large capacitive loads can be applied either to CMOS or to nMOS designs. However, it is nMOS technology that benefits most from a better way to drive signals to 1 quickly, and that lends itself most readily to a set of elegantly simple clock-powered bootstrap circuits. An nMOS circuit that serves as a good approximation of the behavior idealized in Figure 2 is the "Clock-AND":

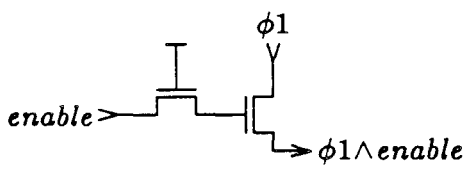


Figure 5

The transistor whose gate is connected to V_{dd} is called the *isolation transistor*, and the transistor that passes the clock signal is called the *clock-pass transistor*. The *enable* signal is assumed to change only during $\phi 2$. The origin of this circuit is probably as old as MOS technology. The earliest reference that we know of that describes the circuit carefully [3] is from 1972, and includes an analysis of a *p*MOS version of the clock-AND.

If *enable* is 0 during $\phi 1$, the isolation transistor holds the gate of the clock-pass transistor at 0, the clock-pass transistor is off, and $\phi 1$ does not pass to the output. If *enable* is at the logic-1 voltage, nominally V_{dd}, the voltage on the gate of the clock-pass transistor just before the beginning of $\phi 1$ is $V_{dd} - V_T$. Then when $\phi 1 : 0 \rightarrow V_c$, the gate of the clock-pass transistor is "bootstrapped" to a voltage more positive than V_c by the coupling of the gate-to-channel capacitance of the clock-pass transistor. The isolation transistor fulfills the role suggested by its name by remaining off while the gate of the clock-pass transistor, also called the bootstrap node, is at a voltage in excess of $V_{dd} - V_T$. The clock voltage, V_c, whether larger or smaller than V_{dd}, is passed to the output. When $\phi 1 : V_c \rightarrow 0$, the output is discharged through the clock-pass transistor back into the clock signal.

The switch-level simulator MOSSIM II [1] models the behavior of these circuits correctly, and we believe that other switch-level simulators can be adapted similarly. However, the bootstrap action and the drive capability of the clock-pass transistor depend on proper transistor sizing.

Although there is no absolute criterion for what is satisfactory bootstrap voltage, this elementary clock-AND is almost foolproof. The capacitance between the gate and channel of the clock-pass transistor gives adequate bootstrap voltage, so long as the parasitic capacitance of the bootstrap node is reasonably minimized in layout. If the parasitic capacitance is too large, the capacitive divider formed between the gate-to-channel capacitance and parasitic capacitances yields too small a V_{gs} on the clock-pass transistor for good bootstrap performance.

For the elementary clock-AND we have used "Speck's rule", which states that the bootstrap action is satisfactory as long as the gate capacitance of the clock-pass transistor is at least 4 times the parasitic capacitance on the bootstrap node, each calculated using typical process parameters. This rule of thumb is based on SPICE simulations with a wide variation of process parameters around those observed for typical MOSIS *n*MOS runs, as well as test chips and projects fabricated on every run since mid-1982.

The rules for selecting the size of the clock-pass transistor are less empirical, and exhibit an interesting speed-power tradeoff. As indicated

in the previous section, the case of interest is when RC, the approximate delay of the clock-AND, is much less than the rise- or fall-time, t_r or t_f. In a typical example from one of our designs, we might assume that $t_r, t_f \geq 5$ns, and we choose the size of the clock-pass transistors to assure that $RC \leq 0.5$ns. In order to drive 0.5 pF of capacitance – 32 minimum geometry transistors and 800μm of poly wire in a 3μm nMOS process (a select line in a RAM) –, R must be less than $1K\Omega$. The voltage across the transistor is small, and the analysis is approximate, so let us use a simple resistive model for the transistor: $1/R = \frac{W}{L}\mu C_{ox}(V_{gs} - V_T)$. For $\mu C_{ox} \approx 50\mu\text{A/V}^2$ and $(V_{gs} - V_T) \approx 2\text{V}^*$, the transistor is about $10K\Omega/\square$, and $\frac{W}{L} = 10$ will serve. This choice results in quite a small area, energy, and delay for "driving" such a load. One is reminded, however, that the clock-AND circuit is not properly called a "driver". It does not provide any power amplification of the clock input.

Having selected the size of the clock-pass transistors in a design to a given limit on RC, *eg* $RC \leq 0.5$ns, one may still trade speed and power dissipation by the choice of t_r, t_f and clock period. The target value of RC determines also the tolerable resistance of the clock distribution conductors.

One caution about using the clock-AND circuit is that when the *enable* signal is 0, the output is floating, and is thus susceptible to charge sharing and spurious capacitive coupling. What is more, unless *enable* is 1 at some regular interval, leakage currents may cause the floating output node to drift to some voltage significantly above 0V. Thus the clock-AND is usually augmented with a *keeper* transistor that keeps the output at ground, such as:

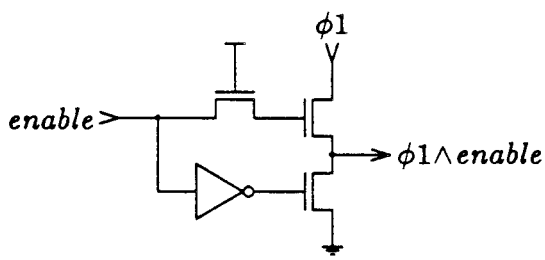


Figure 6

In this circuit the output is always driven. However, this property is not always desirable, and the circuit requires a static inverter, so one

* Body effect increases V_T at the elevated source potential, thus this conservative estimate.

may prefer to drive the keeper with the alternate clock. The output is then kept at 0V during $\phi 2$, so that leakage currents may not accumulate charge on the output node, and is either floating or driven to V_c during $\phi 1$. Such circuits can be used in "wired-OR" forms, and also, clock-ANDs can be put in series with the output of one driving the clock input of another to form an AND function:

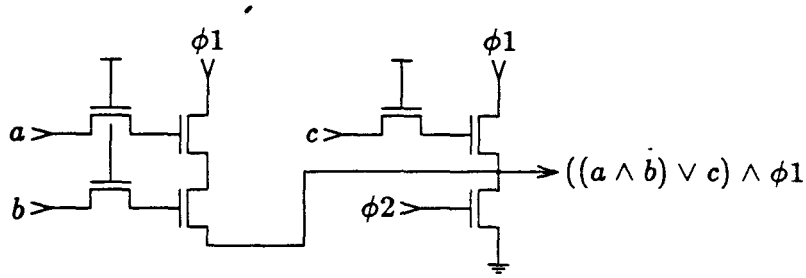


Figure 7

A variant on the clocked keeper is suggested when the ground is not conveniently available in the layout. Since $\phi 1$ is 0 while $\phi 2$ is at V_c, one can as well cause the keeper transistor to sink the output to $\phi 1$, resulting in the following peculiar-looking but perfectly functional circuit:

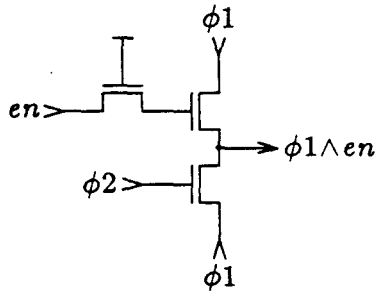


Figure 8

All the work in switching the output in these circuits is done by the clock-pass transistor. The keeper transistor turns on only when the output is already at 0V, and can be of minimum size.

The techniques outlined in this section can be used in conjunction with Mead-Conway style nMOS designs with no known difficulties, even for beginning students. When the students in the Caltech VLSI design course learn about nMOS (following their first project, which is done in CMOS/SOS), they learn these techniques from the start. The resulting

project chips have advantages in (1) higher speed, which in nMOS is otherwise seriously limited by the ability to drive signals to 1 quickly with depletion pullups or "superbuffers", (2) less sensitivity of speed to variations of the depletion transistor threshold, (3) less power dissipation, and (4) less area.

Since the clock signal originates off-chip, it is not strictly necessary that the clock HIGH voltage, V_c, be the same as V_{dd}. There are advantages to making V_c exceed the nominal logic-1 voltage, V_{dd}, by at least the V_T of the enhancement transistors, so that logic-1 voltages are not degraded through pass transistors whose gates are driven by a clock or gated clock. We typically use $V_c = 7\text{V}$ and $V_{dd} = 5\text{V}$, but the circuits also work correctly, although slower and at lower power, with $V_c = 6\text{V}$ and $V_{dd} = 4\text{V}$. Since the saturation current in pass transistors depends on $(V_{gs} - V_T)^2$, a little extra clock voltage also goes a long way to increasing performance. If $V_c = V_{dd}$, the usual ratio rules apply, but if $V_c \geq V_{dd} + V_T$, one can avoid the degraded logic-1 signals that require higher ratios. In addition to passing the V_c voltage, the clock-AND output switches all the way to 0V, a property that is critically important to driving select lines in dynamic RAMs [2].

3. Fancy Circuits

If this hot-clock style of nMOS design – typified by powering circuits from the clock signals – were "carried to its limits", can it also be made universal? That is, can extensions of these circuits be used to implement arbitrary logical functions and reliable clocked storage elements? They can, and we believe to excellent advantage; however, *Fair Warning:* At this point we shall depart from circuits that are reasonably foolproof. Correct operation may depend on capacitance ratios, and charge sharing between nodes of different capacitance may be used determine the direction of signal flow. The tolerance of these circuits to process variation should be checked by circuit simulation.

Let us start with a clocked storage element. Having said so far that the clock-AND *enable* signal changes during $\phi 2$ and remains stable during $\phi 1$, we can make a circuit that latches the control input on $\phi 2$, what we call a *clocked-isolation* clock-AND (Figure 9). This circuit does not require V_{dd}, so this is a good time to dispense with ground as well (except for the substrate), as in the circuit of Figure 8. However, in this or in some of the following circuits, certain clock inputs can be replaced with ground.

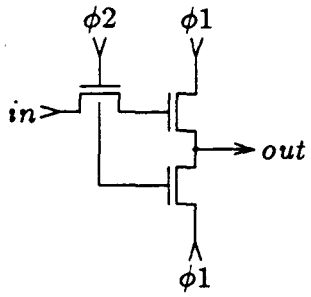


Figure 9

One thing to notice about this circuit is that the isolation transistor can turn on while there is voltage across it, and accordingly, it dissipates power in charging or discharging the bootstrap node. The goal of exporting *all* of the dynamic power is elusive.

When the clocked-isolation clock-AND is used as a clocked storage element, say to make a shift register:

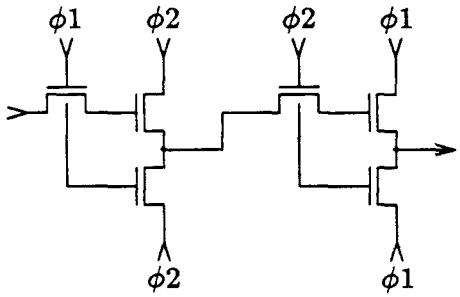


Figure 10

there are several possible modes of misoperation. First of all, charge sharing occurs at the beginning of $\phi 2$ between a floating 0 output of the first stage and the storage/bootstrap node of the second stage. For a typical shift register layout the charge shares in the direction opposite to our intent. Hence this circuit is used only in cases in which the parasitic capacitance between the stages is large. A reliable shift register circuit will be shown after these preliminaries.

Another problem is that a race occurs when the output of the first stage is switching $1 \rightarrow 0$ at the same time as the isolation transistor at the input of the second stage is turning off. The outcome of the race, based on circuit rather than switching considerations, is as we intend,

at least in the absence of clock skew. The skew tolerance of ordinary two-phase clocked circuits is determined by the t_{12} and t_{21} non-overlap periods (§7.2 in [5]). In these hot-clock circuits the skew tolerance is determined also by the relationship between the clock slope, $\frac{V_c}{t_f}$, and the V_T of the isolation transistor. Starting during $\phi 2$ with the output from the first stage equal to V_c, the bootstrap node in the second stage is at a voltage $V_c - V_T$. As $\phi 2$ switches $V_c \rightarrow 0$, the signal at the input to the second stage would have to precede $\phi 2$ by V_T *in voltage* to keep the clocked-isolation transistor on, so that some charge on the bootstrap node could escape. This voltage margin corresponds through the clock slope to a skew margin of $\frac{V_T}{V_c} t_f$ *in time*.

The most difficult problem with the clocked-isolation clock-AND is the case in which the output pulse is not to appear. When the bootstrap node is discharged and – unlike the elementary clock-AND – *isolated* prior to $\phi 1 : 0 \rightarrow V_c$, the clock transition can couple through the gate-drain overlap capacitance of the clock-pass transistor to turn the clock-pass transistor on. The non-linearity in gate-to-channel capacitance is in our favor: when the clock-pass transistor is off, the capacitance from gate to drain is small, but there must be sufficient capacitance on the bootstrap node to absorb the charge coupled from the drain overlap capacitance. We have had some MOSIS circuits work correctly without placing any minimum on the capacitance on the bootstrap node, but only from runs in which the drain overlap capacitance is unusually small. Reliable operation over the range of parameters for MOSIS *n*MOS runs appears to require a capacitance on the bootstrap node at 0 voltage comparable to the "on" capacitance of the clock-pass transistor. A depletion transistor with its gate at 0 volts and source and drain connected to the bootstrap node is a preferred way to provide this capacitance. At 0 volts one has the benefit of the gate capacitance of the "on" transistor, while at voltages more positive than $-V_T$ of the depletion transistor, the parasitic capacitance is minimized.

This clocked-isolation clock-AND is not used very often in our hot-clock designs. However, it exhibits several of the pitfalls that can appear in the extreme form of the hot-clock style; thus the extended discussion of this 3-transistor circuit. Just to complete the story, so that one might understand that this circuit serves as an non-linear amplifier with respect to the input (necessary for level restoration in digital systems), we display in Figure 11 the experimental transfer characteristic of an elementary clock-AND.

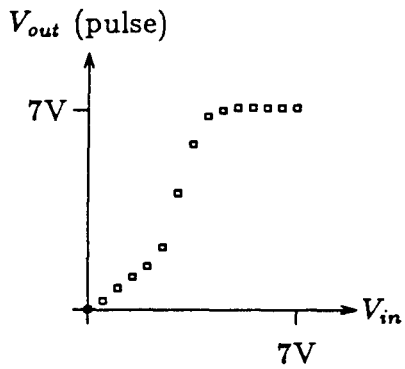


Figure 11

Given the clocked-isolation clock-AND as a clocked storage element and restoring amplifier, it is possible to implement combinational logic by pass networks, such as:

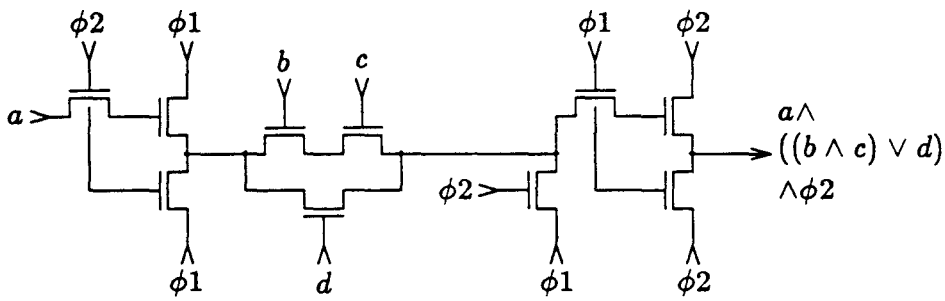


Figure 12

The signals b, c, d may either be stable during $\phi 1$, or may be gated $\phi 1$ pulses. Internal circuit nodes in the pass network capable of significant charge sharing must be pre-discharged during $\phi 2$. A circuit in this form was used, for example, to rotate a quaternary (1-of-4) coded pass input according to a quaternary control input to perform a fast quaternary addition in a multiplier chip.

A pass network demonstrates the ability to compute $\wedge$–$\vee$ expressions, but not complements. It is no great trick to compute the complement of a signal by precharging a node, then discharging it conditional on a clock signal on the following phase:

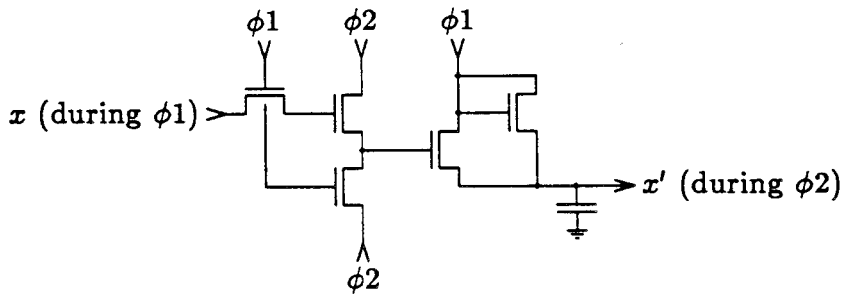


Figure 13

We call the transistor whose gate and drain are both connected to $\phi 1$ a *diode precharger*. It is a pass precharging device that dissipates power only because of its V_T "forward drop". However, the energy stored during $\phi 1$ on the output load capacitance by this diode precharger is (horrors!) *dissipated* in the *discharge transistor* during $\phi 2$ if the output is to be 0. This approach is obviously not very nice if C is large and we are serious about saving dynamic power. Also, the complement is delayed.

The *inverting* clock-AND is based on this same precharge and discharge trick, but more elegantly applied:

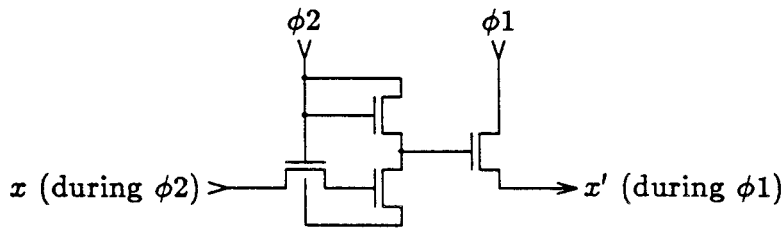


Figure 14

The bootstrap node is precharged by the diode during $\phi 2$. Thus the clock-pass transistor actively sinks the output to $\phi 1$ during $\phi 2$. It is doing double-duty as the keeper (*cf* Figures 8 & 9). The computation of the complement is fairly easy to understand if you first appreciate that it is accomplished in the non-overlap period between $\phi 2$ and $\phi 1$. If the input latched at the end of $\phi 2$ is 0, the bootstrap node remains charged through the next $\phi 1$ epoch, and a $\phi 1$ pulse is produced at the output. If the input latched is 1, the bootstrap node is discharged as $\phi 2$ switches to 0, and no output pulse is produced during $\phi 1$.

Both of the transistors that drive the bootstrap node act as pass devices, and hence dissipate power only due to their V_T drop. The input pass transistor, like the isolation transistor (Fig 9), may switch with voltage across it, but in this circuit the capacitive load may be made smaller than that of the clock-pass transistor. (However, the capacitance of this storage node may need to be augmented since one is trying to store charge on the gate of a transistor while the source and drain are both at V_c, and to retain the charge after the source and drain switch to 0.) Another piece of good news about this circuit is that – unlike the clocked-isolation clock-AND – the bootstrap node is actively held at 0V during $\phi 1$ when the output pulse is not to appear. We need have no anxiety that some process parameter variation will allow the output pulse to appear when we do not mean for it to.

The only possible pitfall in using this circuit is that the output is not actively driven to 0 during $\phi 1$. Thus if there is a potential for charge sharing of a 0 output during $\phi 1$, one may add a pulldown network to the circuit in Figure 14:

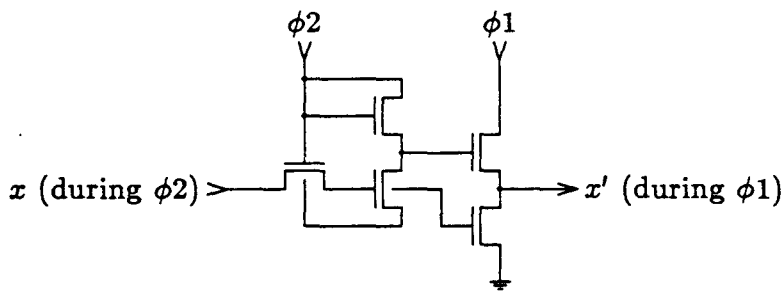


Figure 15

If ground is available on the chip, it should be used as shown in order to augment the storage capacitance. Otherwise, since the clock-pass transistor provides a path to ground during $\phi 2$, a path to ground during $\phi 1$ can be provided on the drain of an extra transistor whose gate is connected to $\phi 1$ and whose source is connected to $\phi 2$. In the form shown in Figure 15, the circuit output is always driven, and the circuit can be composed safely with itself. The same technique producing a complement by diode-charging on one clock phase, with a conditional discharge during the non-overlap period, can also be used to drive the pulldown network. Thus a non-inverting clocked-isolation clock-AND can be made to have this same property of the output being always driven:

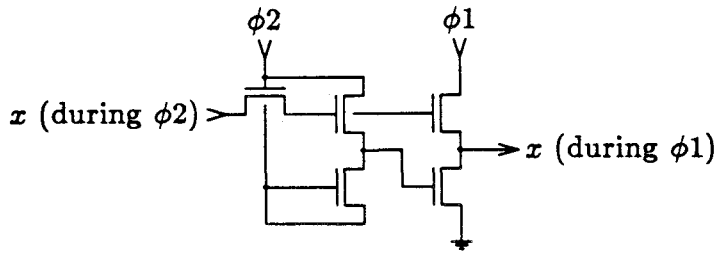


Figure 16

Here again, ground may be used if available, or a path to ground can be generated in a variety of ways from the clock signals.

The discovery of progressively more devious hot-clock nMOS circuits goes on and on, but this paper does not. The interested reader will have no difficulty, and perhaps even some fun, inventing more. Consider, for example, the following final fancy hot-clock circuit, an elaboration on the inverting clock-AND to compute not just the complement, but any switching expression that is negative in all of its inputs:

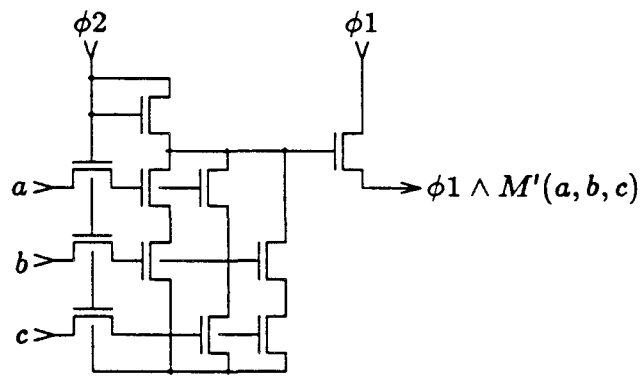


Figure 17

How might the output of this circuit be made to provide an active path to 0 during $\phi 1$? This switching element, while it provides both storage and logic functions, is not (quite) universal by itself, because the functional dependence on an input that enters the network on, say, $\phi 2$ is always negative on $\phi 1$ outputs and positive on $\phi 2$ outputs. For inputs a, b that enter in the same phase, it would not be possible to compute a switching function that is positive in one variable and negative in the other, such as $a \wedge b'$. The combination of inverting and non-inverting clocked-isolation

clock-ANDs is formally universal, and in combination with pass networks has proved to be an effective design paradigm.

4. Applications and Conclusions

MOSIS has fabricated about two hundred Mosaic processors [4], in fact, MOSIS people have used layouts we have assembled for them over a range of feature sizes for yield characterizations of different processes. These processors use the elementary clock-AND circuits extensively; there is hardly a control signal to be found in the Mosaic processor that is not driven by a clock-AND. The observed yields on these processors are indistinguishable from those of other chips of comparable complexity and design refinement. Our students use them with similar results. They have been used successfully in the Pixel-Planes chips [6]. Thus we regard the elementary clock-AND as fully qualified for MOSIS *n*MOS processes.

The Mosaic processors fabricated on 3μm MOSIS runs operate at a clock frequency of about 18 MHz, apparently limited by the pad frame and test jig. One clock period in this machine includes, in parallel, one storage cycle and one microcode cycle, while the datapath performs sequentially an arithmetic operation and a bus transfer. Thus we regard this performance as quite respectable for the technology used. We have observed little variation in performance in the chips from one MOSIS run to another. In designs that use depletion pullups in critical paths we observe the usual sensitivity of speed to the depletion threshold variations.

A array multiplier that uses a quaternary number representation internally was the first of our chips to be designed entirely without depletion devices, and entirely clock-powered. Circuit simulations of the critical paths predicted a 10 MHz throughput, but a complete chip has not yet been fabricated.

The Mosaic RAM, which is described briefly in [4], was the next chip to apply this technique to the limits, indeed, to limits that at the time of its design exceeded our understanding of the tolerance of these circuits to variations in process parameters. This chip includes instances of all of the "fancy" circuits described in section 3. We did, however, receive working silicon for this RAM from a MOSIS run with a fairly large t_{ox} of 65 nm and a very large V_T of 1.2V. Earlier we had received one run of chips that could read but not write, and from SPICE simulations and from studying the misbehavior had isolated the problem to three different clocked-isolation clock-ANDs (figure 9) that were producing an output pulse with a 0 input. Additional capacitance on the bootstrap node is expected to adjust the input switching threshold on these circuits to

provide a tolerance to variation in the ratio of gate capacitance to drain overlap capacitance over about a 2.5 : 1 range.

SPICE simulations of the critical paths of the Mosaic RAM predict operation at 3μm feature size in excess of 20 MHz, and at an on-chip dissipation level of a few milliwatts per 4K-bit section. However, the chips from the one "successful" 4μm run operate only up to 7 MHz, a result that we would like to attribute to marginal operation of certain circuits.

We can readily envision a future *n*MOS process, simplified by the omission of depletion transistors, augmented with additional metal layers, and perhaps including zero-threshold transistors, that would give CMOS a good run for its money. Such a technology used together with resonant clock drivers would provide a substantial gain in the relationship between performance and power dissipation. We believe that such a technology and the hot-clock design style would be particularly suitable for memories, computational arrays, microcomputer arrays, and other structures based on repetitions of one complex chip type (§III.B in [7]).

5. Acknowledgments

We have enjoyed and benefitted from many interesting discussions about "hot-clocking" with our Caltech colleagues Alain J Martin and Richard P Feynman. Special thanks to TEXperts Calvin Jackson and Wen-king Su for their help and nifty systems.

6. References

[1] Randal E Bryant, "A Switch-Level Model and Simulator for MOS Digital Systems", *IEEE TC*, vol C-33, no 2, pp 160-177, February 1984.

[2] James J Cherry, Gerald L Roylance, "A One Transistor RAM for MPC Projects", *Proc Second Caltech Conf on VLSI*, pp 329-341, Caltech Computer Science, January 1981.

[3] Reuben E Joynson, Joseph L Mundy, James F Burgess, Constantine Neugebauer, "Eliminating Threshold Losses in MOS Circuits by Bootstrapping Using Varactor Coupling", *IEEE JSSC*, vol SC-7, pp 217-224, June 1972.

[4] Chris Lutz, Steve Rabin, Chuck Seitz, Don Speck, "Design of the Mosaic Element", *Proc MIT Conf Advanced Research in VLSI*, pp 1-10, Artech Books, 1984.

[5] Carver A Mead, Lynn A Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[6] John Poulton et al, "Implementation of a Full Scale Pixel-Planes System", *Proc 1985 Chapel Hill Conf on VLSI*, Computer Science Press, 1985, (this volume).

[7] Charles L Seitz, "Concurrent VLSI Architectures", *IEEE TC*, vol C-33, no 12, pp 1247-1265, December 1984.

[8] Clark D Thompson, "A complexity theory for VLSI", Dept of Computer Science, Carnegie-Mellon University, Technical Report CMU-CS-80-140, August 1980.

The Design of a Self-timed Circuit for Distributed Mutual Exclusion

Alain J. Martin
Department of Computer Science
California Institute of Technology
Pasadena CA 91125

1. Introduction

The purpose of this paper is twofold: to describe a design method for self-timed systems, and, as an illustration of the use of the method, to derive a self-timed circuit for distributed mutual exclusion.

The method consists in "compiling" a high-level description of the solution—i.e. a set of communicating parallel processes—into a circuit—i.e. a network of elementary operators (*and*, *or*, *C*-element, arbiter, and flip-flop). The compilation is systematic and essentially relies upon the four-phase handshaking expansions of the communication actions. The program of each process is compiled into a set of production rules from which all sequencing has been removed. By matching these "production rules" to those describing the semantics of each operator, the programs are identified with networks of operators.

In order to convince the reader that the method presented is applicable to other than trivial examples, we have chosen to illustrate it with a difficult and relevant problem. We believe that the circuit derived by applying the method presents all aspects of a "good" (and new!) solution. The specification of the circuit is the following: An arbitrary number (> 1) of cyclic automata, called "masters", make independent requests for exclusive access to a shared resource. The circuit should handle the requests from the masters in such a way that

1) any request is eventually granted
2) there is at most one master using the shared resource at any time.

The masters are independent of each other: they don't communicate with each other, and the activity of a master not using the resource should

not influence the activity of other masters.

The circuit should be as *self-timed* or *speed-independent* (C. L. Seitz [6]) as possible. By definition, a self-timed circuit is partitioned into *isochronic* (or *equipotential*) regions: Inside an isochronic region, communication along a wire is instantaneous; communication between isochronic regions can take an arbitrary amount of time. The smaller the size of the isochronic regions relative to the size of the circuit, the more speed-independent the circuit. The circuit should be distributed: it should be a collection of "servers"—one per master—communicating with each other by handshaking (no shared data, and of course no shared clock). Further, no activity should be going on in the circuit when there is no request for the shared resource.

The relevance of the mutual exclusion problem in systems exhibiting concurrency need not be advocated any longer. As we shall see, arbitration among asynchronous signals (i.e. choice of one out of several) is an important part of the mutual exclusion algorithm. Since the metastable properties of arbiters ([1], [6]) make it impossible to put a limit on the amount of time an arbiter takes to reach a stable state, the self-timed character of the solution is a great advantage. Moreover, both the self-timed and distributed properties of the solution facilitate its inclusion into distributed systems.

2. The distributed mutual exclusion algorithm

A master M communicates with its private server m. When M wants to use the shared resource —M is said to be *candidate*— it issues a request to m. When the request is accepted, M uses the resource (for a finite period of time), then informs m that the resource is free again.

The servers are connected in a ring. At any time exactly one (arbitrary) server holds a "privilege". Only the "privileged server" may grant the resource to its master thereby guaranteeing mutual exclusion on the access to the resource. A non-privileged server transmits a request from its master—or from its left-hand neighbor—to its right-hand neighbor. A request circulates to the right (clockwise) until either it reaches a server whose master is candidate (this server ignores the request until it has served its master) or it reaches the privileged server. The privileged server reflects the privilege to the left (counter-clockwise) until it reaches the server that generated the request. This server then becomes privileged, and may grant the resource to its master. The strategy consisting of passing requests clockwise and reflecting the privilege counterclockwise has two important advantages. First, no Boolean message need actually be transmitted. Second, no message need be reflected: the completion of a pending request is interpreted as passing the privilege.

The "high-level" description of the solution is a set of communicating concurrent processes. This is the reference description: it is this description that must be proved correct. All other descriptions that will be generated at the different steps of the "compiling" procedure are automatically correct since each is derived from the previous one by semantics-preserving transformations.

2.1 The programming notation

For the sequential part of the algorithm, we use a subset of E. W. Dijkstra's guarded command language [2], with a simpler syntax. Since it is not the purpose of this paper to deal with formal semantics and correctness proofs, we give only a very informal definition of the semantics of the constructs used.

i) $b\uparrow$ stands for $b := \mathbf{true}$, $b\downarrow$ stands for $b := \mathbf{false}$.

ii) The execution of the statement $[G1 \rightarrow S1 \mid G2 \rightarrow S2]$, where $G1$ and $G2$ are Boolean expressions, and $S1$ and $S2$ are arbitrary program parts, ($G1$ is called a "guard", and $G1 \rightarrow S1$ a "guarded command") amounts to

- the execution of $S1$ if $G1$ holds, or the execution of $S2$ if $G2$ holds (if $G1 \wedge G2$ holds, an arbitrary choice is made between $S1$ and $S2$).
- if $\neg(G1 \vee G2)$ holds, the execution of the statement is suspended until $G1 \vee G2$ holds.

iii) $*[S]$ stands for "repeat S forever".

iv) $[G]$ where G is a Boolean, stands for $[G \rightarrow \mathbf{skip}]$, and thus for "wait until G holds". (Hence, "$[G]$; S" and $[G \rightarrow S]$ are equivalent.)

v) From ii) and iii), the operational description of the statement $*[[G_1 \rightarrow S_1 \mid \ldots \mid G_i \rightarrow S_i \mid \ldots]]$ is "repeat forever: wait until some G_i holds; execute an S_i for which G_i holds".

The implementation of the bar (|) when a non-deterministic choice has to be made is more difficult than when at most one guard is **true**. We shall therefore make the sets of guards as deterministic as possible. One form of non-determinacy is unavoidable when asynchronous signals may change the value of the guards while they are evaluated. This case presents the added complication of the metastability phenomenon, and the bar must then be implemented with an *arbiter*. Since it is difficult (if not impossible) to decide upon the necessity of arbitration at "compile time", we require that the programmer indicate whether a bar is an "arbitration bar" by using another symbol for this case: the thick bar ‖.

Processes communicate with each other by communication actions on channels. A master M communicates with its server m via a channel

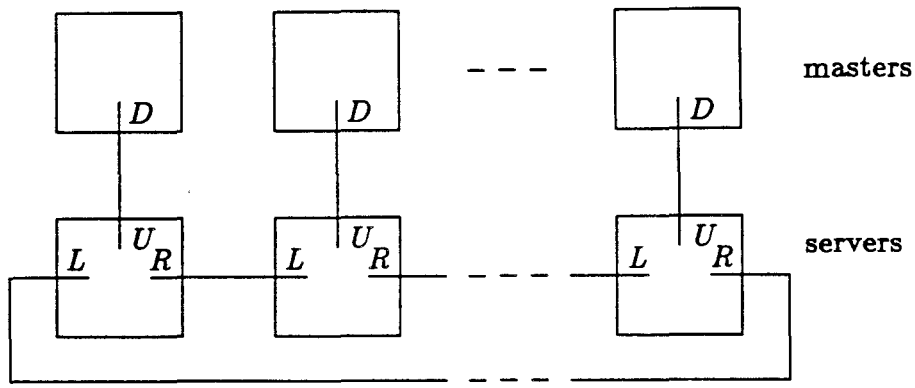


Figure 1

named D in M and U in m. A server m communicates with its right-hand neighbor mr by a channel named R in m and L in mr.

When no messages are transmitted, communication on a channel is reduced to synchronization signals. The name of the channel is then sufficient for identifying a communication action. For instance, the activity of a master relevant to our problem is described as:

$$M \equiv *[\ldots D;\ \text{“use resource”};\ D \ldots].$$

If two processes p and q share a channel named X in p and Y in q, at any time, the number of completed X-actions in p equals the number of completed Y-actions in q. In other words, the completion of the n-th X-action “coincides” with the completion of the n-th Y-action. If p reaches the n-th X-action before q reaches the n-th Y-action, the completion of X is suspended until q reaches Y. The X-action is then said to be *pending*. When q reaches Y, Y is *firable*.

A Boolean command on channels is used, called the *probe*. In process p, the probe command $\overline{X}$ means “X is firable” or in other words, a “Y-action is pending in q”. Hence $\overline{X} \rightarrow X$ guarantees that the X-action is not suspended. (For a more rigorous definition of the communication mechanism, see [3].)

2.2 The program

With the above programming notation, we can now give the first description of a server's algorithm. Each server owns a private Boolean b representing the privilege.

$$\begin{aligned} m \equiv *[[&\overline{U} \rightarrow [b \rightarrow \mathbf{skip} \mid \neg b \rightarrow R];\ U;\ U;\ b\uparrow \\ [\!]\,&\overline{L} \rightarrow [b \rightarrow \mathbf{skip} \mid \neg b \rightarrow R];\ L;\ b\downarrow \\ ]]&. \end{aligned}$$

(For a formal treatment of this algorithm and a correctness proof, see [4].)

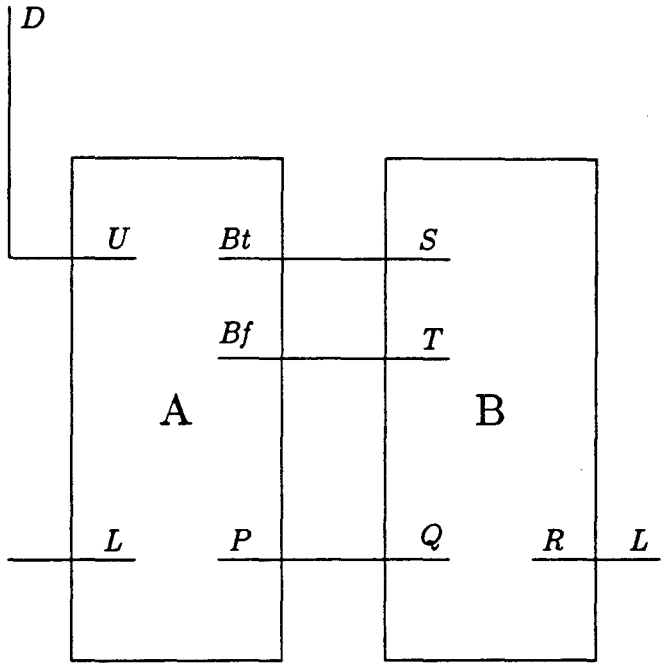


Figure 2

2.3 First decomposition

We distinguish two logical parts in a server's program: the "selector" A, which is the interface with the master and the left neighbor, and the "switch" B, which either transmits a request to the right or reflects it to the left, and which holds and updates the value of b. We therefore decompose process m into two communicating processes A and B. The decomposition is mechanically carried out by applying the following

Decomposition rule: *A process p containing an arbitrary program part* S ($p \equiv \ldots S; \ldots S; \ldots$) *is semantically equivalent to the two processes* $p1$ *and* $p2$, *where* $p1$ *is derived from* p *by replacing each occurrence of* S *by a communication action* C *on the newly introduced channel* C, *and* $p2 \equiv *[[\overline{D} \rightarrow S;\ D]]$ *with* $p2.D = p1.C$.

The communication between A and B is given by Fig. 2.

$$A \equiv *[[\overline{U} \rightarrow P;\ U;\ U;\ Bt \ [\!]\ \overline{L} \rightarrow P;\ L;\ Bf \ ]]\,.$$

$$B \equiv *[[\overline{Q} \wedge b \rightarrow Q \ |\overline{Q} \wedge \neg b \rightarrow R;\ Q \ |\overline{S} \rightarrow b\uparrow;\ S \ |\overline{T} \rightarrow b\downarrow;\ T \ ]]\,.$$

3. The "object code": operators and wires

3.1 Variables and wires

The object code is a collection of Boolean variables and operators

on those variables. A variable belongs exactly to one isochronic region. Because of the definition of isochronic regions, a wire inside an isochronic region represents a variable. Hence the symbols

inside an isochronic region mean $x \equiv y$, and $x \equiv y \wedge y \equiv z$ respectively. In the first case x and y, in the second case, x, y, and z are different names for the same variable. On the other hand, a wire between two isochronic regions—called a *non-isochronic wire*—is an operator. The main assumption on the physical behavior of a non-isochronic wire is the

Monotonicity property: *A monotonic change at one end of a non-isochronic wire is followed, if left undisturbed long enough, by the same monotonic change at the other end of the wire.*

Because we make no assumption on the timing behavior or on the length of non-isochronic wire, in general we cannot guarantee that two consecutive changes at one end of a non-isochronic wire are followed by two corresponding changes at the other end of the same wire. In order to guarantee that all changes at one end of a wire are followed by the corresponding changes at the other end of the wire, we will restrict the use of non-isochronic wires to so-called four-phase handshaking protocols, i.e. as implementations of communication actions.

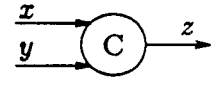
3.2 Operators

Apart from the wire-operator that will be described after introducing the handshaking protocol, the operators used are:

The C-element:

$$(x,y)\ \underline{C}\ z \equiv *[[x \wedge y \rightarrow z\uparrow \mid \neg x \wedge \neg y \rightarrow z\downarrow]].$$

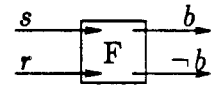
represented as



The set-reset flip-flop:

$$(s,r)\ \underline{F}\ b \equiv *[[s \rightarrow b\uparrow \mid r \rightarrow b\downarrow]].$$

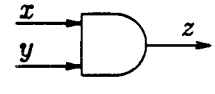
represented as



The "and":

$$(x,y)\ \underline{\wedge}\ z \equiv *[[x \wedge y \rightarrow z\uparrow \mid \neg x \vee \neg y \rightarrow z\downarrow]].$$

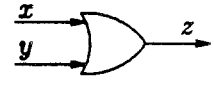
represented as



The "or":

$$(x,y)\ \underline{\vee}\ z \equiv *[[x \vee y \rightarrow z\uparrow \mid \neg x \wedge \neg y \rightarrow z\downarrow]].$$

represented as



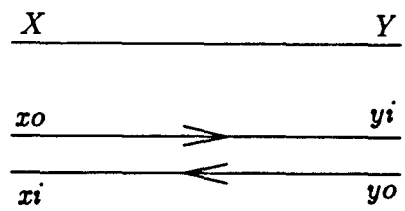
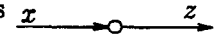


Figure 3

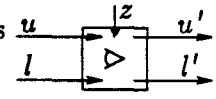
The inverter:

$$x \sqsupset z \equiv *[[x \rightarrow z\downarrow \;|\; \neg x \rightarrow z\uparrow]].$$

represented as 

The arbiter:

$$(u,\ell,z)\ \underline{A}\ (u',\ell') \equiv *[[u \wedge \neg\ell' \wedge \neg z \rightarrow u'\uparrow \;|\; \ell \wedge \neg u' \wedge \neg z \rightarrow \ell'\uparrow \;|\; u' \wedge \neg u \rightarrow u'\downarrow \;|\; \ell' \wedge \neg\ell \rightarrow \ell'\downarrow]].$$

represented as 

In the arbiter, z is an inhibition variable that is used to postpone the firing of one of the first two commands until a sequence of actions caused by the previous firing of one of these two commands, is terminated. *A guarded command is called a production rule when the command contains no sequencing (no semicolon), as is the case for all operators described above.* The arbiter is the only operator able to make an arbitrary choice between two firable production rules (the conditions $u \wedge \neg\ell' \wedge \neg z$ and $\ell \wedge \neg u' \wedge \neg z$ can be **true** at the same time). (We assume that the arbiter makes a **fair** choice between the two commands, i.e. it is excluded that a command becomes firable infinitely often without being selected.) For all other operators, only one production rule may be firable at any time. This is only a problem for the flip-flop for which $\neg(s \wedge r)$ must hold at any time.

4. Handshaking

4.1 Four-phase handshaking

The channel (X,Y) shared by processes p and q is implemented by a pair of "directed wires": it is replaced in p by the output wire xo and the input wire xi, and in q by the output wire yo and the input wire yi (Fig. 3).

A matching pair of communication actions is always implemented as one output action denoted !, and one input action denoted ?. When no message is actually transmitted, an arbitrary choice is made unless the choice is dictated by the presence of probes. The communication actions on (X, Y) can be implemented either as X output ($X!$) and Y input ($Y?$):

$$X! \equiv xo\uparrow;\ [xi];\ xo\downarrow;\ [\neg xi] \tag{1}$$
$$Y? \equiv [yi];\ yo\uparrow;\ [\neg yi];\ yo\downarrow \tag{2}$$

or vice versa ($X?, Y!$). Initially, all variables are **false**. A "probed" communication action $\overline{X} \rightarrow \ldots X$ must be implemented:

$$xi \rightarrow \ldots;\ xo\uparrow;\ [\neg xi];\ xo\downarrow. \tag{3}$$

This implementation of probes forces the matching communication action to be implemented as an output. (Hence the two actions of a matching pair cannot be both probed.)

4.2 Basic properties

Several properties of the above handshaking protocol that play an important role in the compilation method are now given without proof.

Property 1: *The implementation of communication actions described in (1), (2), and (3) fulfills the semantics of communication actions and probe as given in 2.1.*

Property 2: *A wire between two isochronic regions that is used only for implementing a channel as described in 4.1, behaves as a so-called "wire operator" $\underline{W}$. The directed wire from x to y is the operator $x\ \underline{W}\ y$, with the semantics:*

$$x\ \underline{W}\ y \equiv *[[x \rightarrow y\uparrow \mid \neg x \rightarrow y\downarrow]].$$

(In other words, the use of a non-isochronic wire in a handshaking protocol guarantees that any change at the input of the wire is effectively followed by the same change at the output of the wire.)

Property 3: *For the pair of wires $xo\ \underline{W}\ yi$ and $yo\ \underline{W}\ xi$, used together as in (1) and (2), and all variables* **false** *initially, the following sequence of transitions is guaranteed to occur if the system is deadlock-free:*

$$\{xo\uparrow,\ yi\uparrow,\ yo\uparrow,\ xi\uparrow,\ xo\downarrow,\ yi\downarrow,\ yo\downarrow,\ xi\downarrow\}^*.$$

Property 4: *Consider a program p containing a sequence of handshaking actions corresponding to the implementation of a communication action according to (1) or (2). Provided that the cyclic order of the four actions is respected, the last two actions can be inserted at any place in p without invalidating the semantics of the communication involved. However, modifying the order of these two actions relatively to other actions of p may introduce deadlock.*

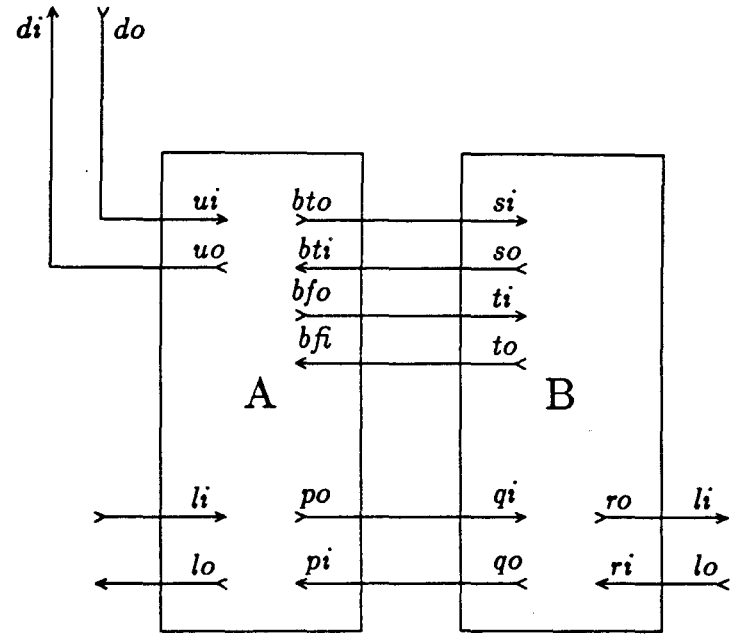


Figure 4

5. The "compilation" procedure

The first step of the "compilation" consists in replacing each communication action by its four-phase handshaking implementation. This compilation step is called the *handshaking expansion* of the program.

5.1 Handshaking expansion

The expansion of the communication structure according to the handshaking protocol of 4.1 is shown in Fig. 4.

In the programs of M, A, and B we are now going to replace the communication actions by their handshaking implementation. When this mechanical task is completed, and possibly after clerical simplifications, some transformations are applied based on Property 4.

We perform a simple optimization on the expansion of A and consequently on the expansion of M: to suppress one of the two consecutive U-expansions. The expansions of M and A become:

$$M \equiv *[\ldots;\ do\uparrow;\ [di];\ \text{"use resource"};\ do\downarrow;\ [\neg di]\ldots]. \tag{4}$$

$$\left.\begin{array}{l} A \equiv *[[ui \rightarrow po\uparrow;\ [pi];\ po\downarrow;\ [\neg pi];\ uo\uparrow;\ [\neg ui]; \\ \qquad\qquad uo\downarrow;\ bto\uparrow;\ [bti];\ bto\downarrow;\ [\neg bti] \\ \quad [\!]\ \ell i \rightarrow po\uparrow;\ [pi];\ po\downarrow;\ [\neg pi];\ \ell o\uparrow;\ [\neg \ell i]; \\ \qquad\qquad \ell o\downarrow;\ bfo\uparrow;\ [bfi];\ bfo\downarrow;\ [\neg bfi] \\ ]]. \end{array}\right\} \tag{5}$$

The straightforward expansion of B gives:

$$
\left.
\begin{array}{l}
B \equiv *[[qi \wedge b \rightarrow qo\uparrow;\ [\neg qi];\ qo\downarrow \\
\quad |qi \wedge \neg b \rightarrow ro\uparrow;\ [ri];\ ro\downarrow;\ [\neg ri];\ qo\uparrow;\ [\neg qi];\ qo\downarrow \\
\quad |si \rightarrow b\uparrow;\ so\uparrow;\ [\neg si];\ so\downarrow \\
\quad |ti \rightarrow b\downarrow;\ to\uparrow;\ [\neg ti];\ to\downarrow \\
\quad]].
\end{array}
\right\} (6)
$$

5.2 Production rules expansion

The next step is to compile the handshaking expansions of the programs into sets of production rules from which all explicit sequencing has been removed. By matching those production rules to those describing the semantics of each operator, the programs can be identified with networks of operators. Consider the compilation of B.

5.3 Compilation of B

The right-hand side of each production rule in the next compilation of B will contain exactly one transition of (6), i.e. one of the actions $qo\uparrow$, $qo\downarrow$, $ro\uparrow$, $ro\downarrow$, etc ... Since there are twelve occurrences of these transitions in (6), there will be twelve production rules. The problem is to define the guard of each rule such that the sequence of firings of these rules is equivalent to the execution of (6). In determining these guards we take into account that

i) all handshaking variables are initialized to **false**,

ii) Property 3 induces sequencing among set of transitions,

iii) an upward transition ($x\uparrow$) can be guarded by negated variables only, if it is the first ("spontaneous") transition of a sequence. (Obviously, only the first transition in a master's sequence is of this type. None of the servers' transitions is.)

The production rules for the first, third, and fourth guarded commands of (6) can be derived easily. For instance, for the first guarded command we obtain the rules

$$qi \wedge b \rightarrow qo\uparrow$$
$$\neg qi \wedge b \rightarrow qo\downarrow .$$

But we have some difficulty in determining the guard of $qo\uparrow$ in the second guarded command, since the only variable with value **true** as a precondition of $qo\uparrow$ is qi. In order to define uniquely the state in which the transition $qo\uparrow$ is to take place we can, of course, introduce a state variable. But the implementation of a state variable requires in general a C-element or a flip-flop, and in this solution we choose to minimize the number of those "expensive" state-holding elements.

Instead, we are going to transform the second guarded command according to the transformation allowed by Property 4: we postpone the actions $ro\downarrow;\ [\neg ri]$ until after $[\neg qi]$. The command now becomes:

$$qi \wedge \neg b \rightarrow ro\uparrow;\ [ri];\ qo\uparrow;\ [\neg qi];\ ro\downarrow;\ [\neg ri];\ qo\downarrow\ . \qquad (7)$$

We have to check that the insertion of $qo\uparrow;\ [\neg qi]$ between the two halves of the R-protocol sequence does not introduce deadlock. This is easily done by observing that when B is suspended at $[\neg qi]$, the next action of A is $po\downarrow$ and thus there cannot be deadlock. With this transformation, the compilation of (7) is possible without extra state variables. Hence the complete production rule compilation of B:

$$\begin{aligned}
B \equiv *[[\ & qi \wedge b \rightarrow qo\uparrow && (B1)\\
|\ & \neg qi \wedge b \rightarrow qo\downarrow && (B2)\\
|\ & qi \wedge \neg b \rightarrow ro\uparrow && (B3)\\
|\ & ri \rightarrow qo\uparrow && (B4)\\
|\ & \neg qi \wedge \neg b \rightarrow ro\downarrow && (B5)\\
|\ & \neg ri \rightarrow qo\downarrow && (B6)\\
|\ & si \rightarrow b\uparrow && (B7)\\
|\ & b \wedge si \rightarrow so\uparrow && (B8)\\
|\ & \neg si \rightarrow so\downarrow && (B9)\\
|\ & ti \rightarrow b\downarrow && (B10)\\
|\ & \neg b \wedge ti \rightarrow to\uparrow && (B11)\\
|\ & \neg ti \rightarrow to\downarrow && (B12)\\
]]. &
\end{aligned}$$

We leave it to the reader to check that the execution of the above program started with all handshaking variables false and an arbitrary value of b, corresponds to the execution of (6) started in the same state. (The ordering between handshaking transitions stated in Property 3 has to be taken into account.)

We are now able to implement sets (pairs, actually) of rules with operators on variables of B. $B1$ and $B2$ correspond to $(qi, b) \triangle qo$. $B4$ and $B6$ correspond to $ri = qo$ (isochronic wire or super-buffer). Hence, we implement those four rules as:

$$((qi \wedge b), ri) \underline{\vee}\ qo\ .$$

The implementation of the other rules is straightforward.

$B3$ and $B5$: $(qi, \neg b) \triangle ro$.
$B7$ and $B10$: $(si, ti)\ \underline{F}\ b$, since $\neg(si \wedge ti)$ holds at any time.
$B8$ and $B9$: $(b, si) \triangle so$.
$B11$ and $B12$: $(\neg b, ti) \triangle to$.

This network is represented in Fig. 5.

5.4 Compilation of A

The compilation of A is slightly more complicated since it requires the use of an arbiter. Let us postpone the arbitration issue for a while and concentrate on the production rule implementation of the first guarded

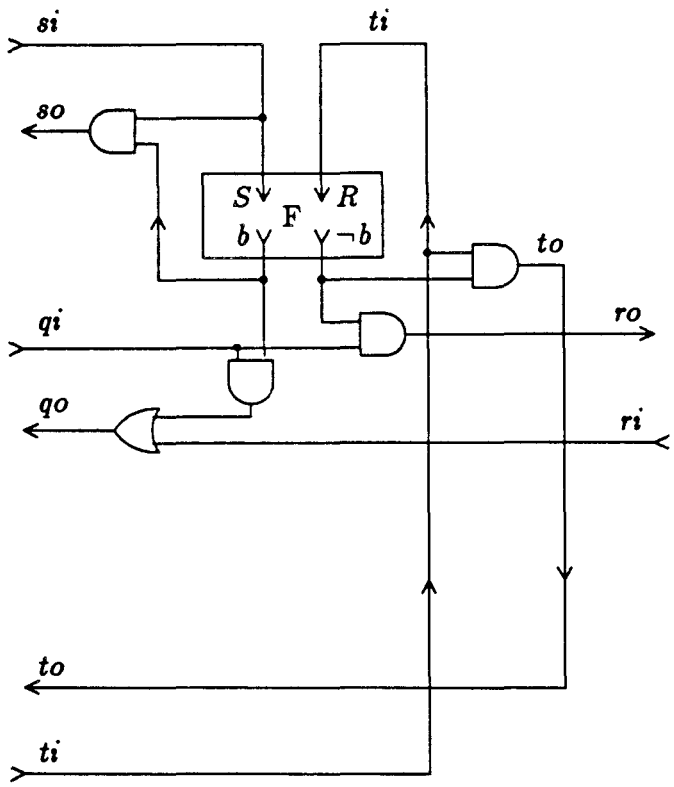


Figure 5

command of (4). (Since the two commands are symmetrical, the implementation of the second one is identical.) We meet the same difficulty for the production rules of the upward transitions $uo\uparrow$ and $bto\uparrow$ as for $qo\uparrow$ in the compilation of B. And we apply the same technique to resolve it: applying the transformation allowed by Property 4, we postpone the actions $po\downarrow$; $[\neg pi]$ until after $[\neg ui]$, and the action $uo\downarrow$ until after $[bti]$. The code of A now becomes:

$$\left.\begin{array}{l} *[[ui \rightarrow po\uparrow;\ [pi];\ uo\uparrow;\ [\neg ui];\ po\downarrow;\ [\neg pi]; \\ \qquad\qquad bto\uparrow;\ [bti];\ uo\downarrow;\ bto\downarrow;\ [\neg bti] \\ \ [\!]\ \ell i \rightarrow po\uparrow;\ [pi];\ \ell o\uparrow;\ [\neg \ell i];\ po\downarrow;\ [\neg pi]; \\ \qquad\qquad bfo\uparrow;\ [bf i];\ \ell o\downarrow;\ bfo\downarrow;\ [\neg bf i] \\ ]]\,. \end{array}\right\} \quad (5')$$

The verification that those transformations do not introduce deadlock is hardly more difficult than in the previous case, and is omitted.

In order to implement the thick bar in (5'), we have to introduce an arbiter $(ui, \ell i, z)\ \underline{A}\ (u', \ell')$. The next coding of A consists of the composition of the guarded commands of the arbiter and the guarded commands of (5') in which u' and ℓ' have been substituted for ui and ℓi:

$$
\left.
\begin{array}{l}
*[[ui \wedge \neg \ell' \wedge \neg z \rightarrow u'\uparrow \\
\quad |\ell i \wedge \neg u' \wedge \neg z \rightarrow \ell'\uparrow \\
\quad |u' \wedge \neg ui \rightarrow u'\downarrow \\
\quad |\ell' \wedge \neg \ell i \rightarrow \ell'\downarrow \\
\quad |u' \rightarrow po\uparrow;\ [pi];\ uo\uparrow;\ [\neg u'];\ po\downarrow; \\
\qquad\qquad [\neg pi];\ bto\uparrow;\ [bti];\ uo\downarrow;\ bto\downarrow;\ [\neg bti] \\
\quad |\ell' \rightarrow po\uparrow;\ [pi];\ \ell o\uparrow;\ [\neg \ell'];\ po\downarrow; \\
\qquad\qquad [\neg pi];\ bfo\uparrow;\ [bfi];\ \ell o\downarrow;\ bfo\downarrow;\ [\neg bfi] \\
]].
\end{array}
\right\} \quad (8)
$$

For the inhibition variable z (z is an internal variable of A), we choose:

$$z = uo \vee \ell o \vee bti \vee bfi\,.$$

With this definition of z, it is easy to verify that the execution of (8) is equivalent to the execution of (5') started in the same state. The transformation of the 5th and 6th guarded commands of (8) into production rules is now straightforward. For the 5th guarded command, we get:

$$
\begin{array}{rl}
u' \rightarrow po\uparrow & (A1) \\
u' \wedge\ pi \rightarrow uo\uparrow & (A2) \\
\neg u' \rightarrow po\downarrow & (A3) \\
\neg pi \wedge uo \rightarrow bto\uparrow & (A4) \\
bti \rightarrow uo\downarrow & (A5) \\
\neg uo \rightarrow bto\downarrow\,. & (A6)
\end{array}
$$

Observe that since $bti \wedge bfi \Rightarrow z$, our choice of the inhibition variable takes care of the implementation of the last transitions $[\neg bti]$ and $[\neg bfi]$. The implementation in terms of operators is as follows.

For $A1$ and $A3$: $u' = po$. For $A4$ and $A6$: $(\neg pi, uo) \triangleq bto$, since $\neg pi$ holds as precondition of $A6$. For $A2$ and $A5$, we strengthen the guards as:

$$
\begin{array}{rl}
u' \wedge pi \wedge \neg bti \rightarrow uo\uparrow & (A2) \\
\neg(u' \wedge\ pi) \wedge bti \rightarrow uo\downarrow, & (A5)
\end{array}
$$

which admits the implementation $((u' \wedge pi), \neg bti)\ \underline{C}\ uo$.

Combining this set of operators with those of the other guards, we obtain the complete set:

$$
\begin{array}{ll}
(ui, \ell i, z)\ \underline{A}\ (u', \ell') & \\
(uo, \ell o, bti, bfi)\ \underline{\vee}\ z & \text{(generalized or)} \\
(u', \ell')\ \underline{\vee}\ po & \\
(\neg pi, uo)\ \underline{\triangle}\ bto & \\
(\neg pi, \ell o)\ \underline{\triangle}\ bfo & \\
((u' \wedge pi), \neg bti)\ \underline{C}\ uo & \\
((\ell' \wedge pi), \neg bfi)\ \underline{C}\ \ell o\,. &
\end{array}
$$

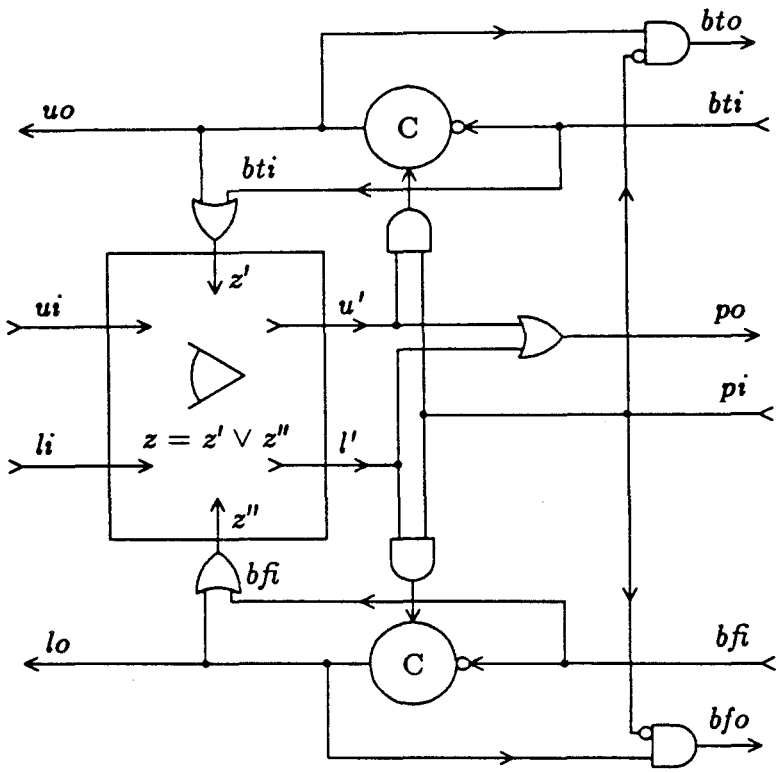


Figure 6

This network is represented in Fig. 6. The complete circuit is represented in Fig. 7.

6. Conclusion

We have described a method for implementing a high-level concurrent algorithm (a set of communicating processes) as a network of digital operators. We have chosen to illustrate the method with an example: the design of a self-timed circuit for distributed mutual exclusion. Although the example chosen is far from trivial, we arrive at the solution by a series of systematic, semantics-preserving, transformations that we have compared to compiling.

The proofs that the transformations preserve the semantics of the algorithms rely on four properties of the four-phase handshaking protocol with which the communication primitives are implemented. Although the proofs of these properties have been omitted, the reader should have no difficulty in being convinced of their correctness, and thus of the correctness of the transformations performed.

The main step in the translation process is the transformation of a strictly sequential algorithm—the handshaking expansion of a communicating process—into a set of production rules from which all explicit

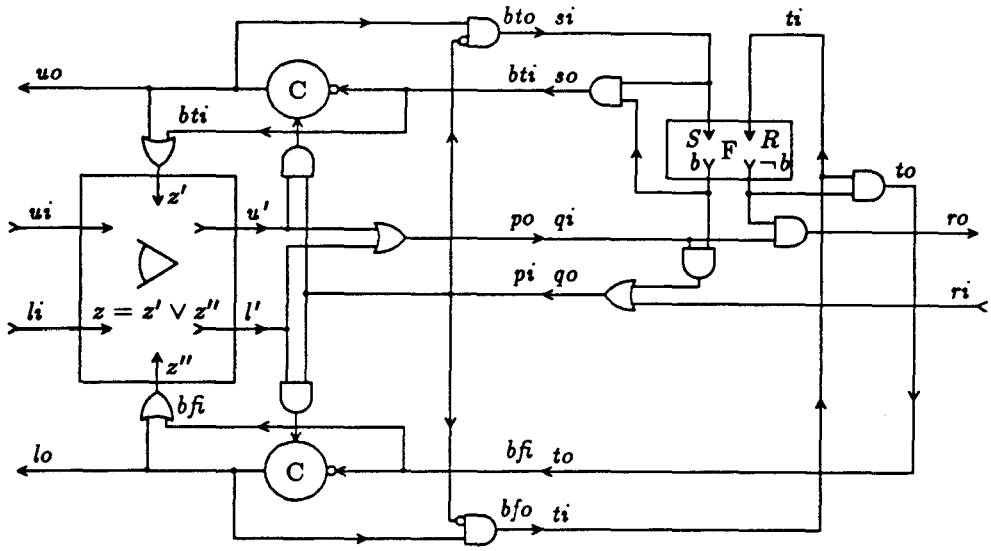


Figure 7

sequencing has been removed. Yet, we guarantee that the rules are fired one at a time in a sequence equivalent to the sequential execution of the original program. We have seen that for the production rule of certain transitions—upward transitions following a complete handshaking sequence—it is impossible, without transformation, to determine a guard that enforces the required sequencing. We have mentioned that this problem can be solved in a quite mechanical way by introducing a state variable that identifies this state uniquely, i.e. by introducing a C-element or a flip–flop.

Although, in this case, the number of extra C-elements thus introduced would be moderate (2 or 3), we aimed at minimizing the number of state-holding elements, and have opted for another—less mechanical—method, consisting of "moving" the second half of some handshaking sequences. This method requires a little care concerning deadlock, but can lead to quite interesting solutions. We believe that, in the solution obtained, the number of state-holding elements is minimal.

Observe that since only one production rule is fired at a time the occurrence of hazards is excluded.

Acknowledgment

Acknowledgment is due to Chuck Seitz for several helpful suggestions on the choice of operators and the structure of arbiters [7] and for invaluable comments on previous versions of the paper, and to Jan van

de Snepscheut and Martin Rem for their comments on a previous solution. Special thanks to our TEXperts Calvin Jackson and Wen-King Su for their crucial help.

The research described in this paper was sponsored in part by the Defense Advanced Research Projects Agency, ARPA Order number 3771, and was monitored by the Office of Naval Research under contract number N00014-79-C-0597.

References

[1] Chaney, T.J. and C.E. Molnar, "Anomalous Behavior of Synchronizer and Arbiter Circuits", *IEEE Transactions on Computers*, vol. C-22, no.4, April 1973, pp. 421-422

[2] Dijkstra, Edsger W., *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs NJ (1976)

[3] Martin, A.J., "The Probe: An Addition to Communication Primitives", to appear in *Information Processing Letters* (1985), also Caltech Computer Science Report 5124:TR:84

[4] Martin, A.J., "Distributed Mutual Exclusion on a Ring of Processes" Computer Science, Caltech, 5080:TR:83

[5] Mead, C. and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading MA (1980)

[6] Seitz, C.L., "System Timing", Chapter 7 in Mead & Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, MA (1980)

[7] Seitz, C.L., Computer Science, Caltech, Lecture Notes (1983)

A Hardware Accelerator for Switch-Level Simulation

William J. Dally
California Institute of Technology
Pasadena, California 91125

Randal Bryant
Carnegi-Mellon University
Pittsburg, Pennsylvania 15213

ABSTRACT. The Mossim Simulation Engine (MSE) is a hardware accelerator for performing switch-level simulation of MOS VLSI circuits. [1,2] Functional partitioning of the MOSSIM algorithm, and specialized circuitry are used by the MSE to achieve a performance improvement of $\approx$ 300 over a VAX 11/780 executing the MOSSIM II program. Several MSE processors can be connected in parallel to achieve additional speedup. A virtual processor mechanism allows the MSE to simulate large circuits with the size of the circuit limited only by the amount of backing store available to hold the circuit description.

KEYWORDS: simulation, switch-level simulation, simulation engines, concurrent architectures

1 Introduction

As the complexity of VLSI circuits approaches 10^6 devices, the computational requirements of design verification are exceeding the capacity of general purpose computers. To provide the computing power required to verify these complex VLSI chips, special purpose hardware for performing simulation is required. Existing logic simulation engines [3-8] are inadequate for MOS VLSI because they cannot accurately model MOS circuits. Switch-level simulation, on the other hand, models the effects of capacitance and transistor ratios at speeds comparable to logic simulation. Existing machines limit the size of a circuit which can be simulated by binding circuit elements to hardware at compile time. Virtual network processing allows circuits of any size to be simulated by binding circuit elements to hardware at run-time.

Design verification plays an essential role in the development of a VLSI chip. The complexity of the circuits, the inaccessibility of internal nodes, and the difficulty of repair make the probability of producing a working chip very low without extensive design verification. The burden of simulation is compounded by the fact that design is an iterative process. Each time a bug is discovered in the circuit and the design modified, all simulations must be repeated to verify that no additional bugs have been introduced by the modification. As circuits become more complex, special purpose simulation hardware is required to perform design verification in reasonable time.

A state of the art VLSI chip in 1982 contained $\approx 10^5$ devices and required about 1 week of CPU time to complete a single verification cycle. Since both the number of test vectors required to verify a chip and the amount of computation required to simulate one test vector scale at least linearly with the

complexity of a chip, the amount of computation required verify a chip at switch level scales at least quadratically with complexity. Thus, as shown in figure 1, a 1986 chip containing $\approx 10^6$ devices will require about 2 years of CPU time to completely verify on a conventional computer. Because of these prohibitive simulation times many groups are abandoning whole chip simulation at the switch-level and moving toward mixed-mode simulation. In the long run both mixed-mode simulation and special purpose hardware will be required to meet the growing demands of VLSI.

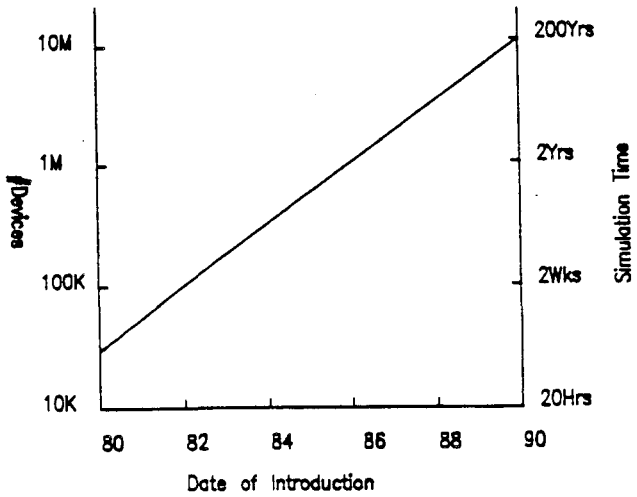


Figure 1. Scaling of Simulation Time

Conventional simulation engines such as the Yorktown Simulation Engine [3-5] and ZyCAD Logic Evaluator [6] address this problem of long simulation times by performing logic simulation orders of magnitude faster than conventional computers. However, these logic simulation engines do not accurately model the behavior of MOS circuits. Simulation of charge sharing, precharged busses, sneak paths and other subtleties of MOS design are beyond the capabilities of these machines. Switch-level simulation is required to model these effects.

Bryant [9-12] has developed MOSSIM, a switch-level model and simulator which provides significantly more accurate simulation of MOS circuits than conventional methods of logic simulation [13]. By modeling MOS transistor ratios and node capacitances and by considering a MOS transistor as a truly bidirectional device, MOSSIM provides simulation capabilities which previously required a circuit simulator [14-17] with simulation times comparable with logic simulation.

This paper describes the architecture of the MOSSIM Simulation Engine (MSE), a special purpose processor for switch-level simulation of MOS circuits. [1,2] The MSE combines the speed advantages of hardware simulation engines with the accuracy of the switch level simulation. The MSE also uses a virtual processor mechanism which allows run-time binding of circuit subnetworks to simulation hardware.

The next section surveys previous work in the field of hardware simulation machines. The conventional simulation engines described suffer both from an inability to accurately model MOS circuits, and from the restrictions of *compile-time* binding of circuit elements and simulation hardware. The MOSSIM algorithm [11] overcomes this first limitation by using a switch-level model of MOS circuits. An adaptation of the MOSSIM algorithm for hardware is described in section 3. Also described in this section is the mechanism of *virtual processors* which allow circuit elements to be assigned to simulation hardware at run-time on a demand basis. Section 4 describes the architecture of the MSE. The architecture is developed by observing how potential concurrency in the MOSSIM algorithm can be exploited by

hardware. Section 5 discusses the performance of the MSE. The paper concludes with a status report on the MSE project and some directions for future research.

2 A Survey of Simulation Machines

Existing logic simulation engines [3-8] reduce simulations times by several orders of magnitude compared to conventional computers. The techniques used to achieve this performance improvement can be applied to switch-level simulation as well. In this section we discuss two logic simulation machines: the Yorktown Simulation Engine (YSE) and the ZyCAD Logic Evaluator (ZLE). We compare these machines with the MSE and identify their strengths and weaknesses.

2.1 The Yorktown Simulation Engine

The YSE is a multiprocessor composed of up to 256 *logic processors* connected by a crossbar switch.[3-5] Each logic processor simulates a logic subnetwork of up to 4K four-input gates. The YSE performs either zero delay or unit delay logic simulation using an approach where every gate is simulated every cycle. This *rank order* or *compiled logic* [18] approach is inefficient as the activity in logic circuits exhibits considerable locality and typically no more than 5%to 10%of the gates are active at any given time. Although a simulation speed of 12.5M gate evaluations per second (GEPS) is claimed for the YSE, because of the sparse nature of logic activity typically no more than 1.25M effective GEPS will be achieved.

The YSE is based on a unidirectional gate model which does not lend itself to MOS switch-level simulation. Although a switch level simulator has been *programmed* on the YSE, by devising logic circuits which model the path strength equations of the switch-level model [11],[19], this approach to switch level simulation suffers from several limitations. First, since several logic gates are required to simulate a single transistor, the approach is inherently inefficient. Few values of strengths are available, making the modeling of phenomena such as charge sharing difficult. Also, the lack of an event-driven simulation kernel makes it impossible to determine when path finding has completed, forcing all path tracing to be run for a maximum time bound.

2.2 The ZyCAD Logic Evaluator

The ZLE is the first commercially available simulation machine.[6] The ZLE consists of up to 15 processors each of which simulates a subnetwork of up to 64K gates. The machine simulates circuits modeled as a network of three-input gates and one-input bidirectional elements. While technical details of the machine have not been published, the available sales literature [6] indicates that it uses *event driven* [18] simulation which overcomes many of the limitations of the YSE's compiled-logic simulation.

The ZLE includes two features which attempt to bridge the gap between logic simulation and switch-level simulation: bidirectional elements and signal strengths. Later models of the ZLE simulate a bidirectional element. This element propagates logic signals between two I/O ports, A and B, with the direction of propagation determined by a control port, C. This bidirectional element has several shortcomings compared to the switch level model. Since the bidirectional element only operates one direction at a time, simulation using this model will not detect sneak paths in a circuit. Since the bidirectional element propagates only a logic signal and does not reflect the electrical characteristics of one port to

the other, charge sharing effects will not be detected. By not propagating strength information through bidirectional elements, the ZLE cannot model complex pass transistor structures such as barrel shifters and multiplexors with all transistors represented as bidirectional elements. Although these structures do not actually exploit the bidirectional characteristics of MOSFET's, an error in the design might cause a sneak path that would go undetected if it were simulated with unidirectional pass transistor models. Also, since the ZLE does not perform path tracing, an unknown on the control input of a bidirectional element leads to an overly pessimistic propagation of unknowns in the circuit.

The ZLE uses three-valued strength system where the three strengths correspond to driven, pulled and charged. While this system allows the user to model ratio logic and wire-OR busses, it falls short of the switch level model in the following areas: Three strengths are usually not sufficient to model the number of different node sizes and transistor strengths in a circuit. The strengths are used only to determine the state of a net and are not propagated through circuit elements.

2.3 Comparison

As shown in Table 1, the two logic simulation machines described above offer better logic simulation speed than the MSE; however this speed is achieved at the expense of accuracy and flexibility. The MSE is the only machine supporting switch-level MOS models and run-time binding of circuit elements to simulation hardware.

	MSE	YSE	ZLE
Speed/Proc (GEPS)	250K	12.5M	2.5M
Gates/Proc	4K	4K	64K
Max Gates	none	1M	1M
Event Driven	yes	no	yes
MOS models	yes	no	no
Run-Time Binding	yes	no	no

Table 1. Comparison of Simulation Engines

2.4 Strengths of Conventional Simulation Engines

Conventional simulation engines are quite successful at improving the performance of logic simulation by orders of magnitude over conventional computers. The techniques used by these machines to achieve speed improvement include: specialization, functional concurrency, subnetwork concurrency and mixed mode simulation. Each of these techniques can be applied to switch-level simulation as well.

Specialization involves dedicating special hardware to perform a frequently occurring operation. An example of specialization is the function unit in the YSE which simulates a general four input logic gate in one cycle. Without this special hardware, several cycles would be required to determine the gate output.

Functional Concurrency is achieved by performing several of the simulation functions in parallel. An example of functional concurrency is the pipelined gate evaluation in the YSE where several functions (fetch instruction, fetch data, simulate gate, store result) are performed simultaneously. The success of

this technique depends on proper balancing of the function units to prevent any one function unit from becoming a bottleneck.

Subcircuit Concurrency is achieved by partitioning the circuit into subcircuits and processing each subcircuit simultaneously. Both of the machines described above make some use of subcircuit concurrency. The low activity of logic circuits acts as a two-edged sword in this case. The locality of the logic activity minimizes inter-processor communication preventing processors from idling due to communications delays. However, since only 5%to 10%of the logic circuits are active at once, and since the active circuits tend to be clustered, only 5%to 10%of the processors may be active at once.

Mixed Mode Simulation: The capability to simulate *uninteresting* subcircuits at a high level gives a significant performance improvement. Much less computation is required to simulate one high-level unit (eg. a 64K RAM) than to simulate many smaller units and their interactions (eg. the 64K+ transistors in the RAM). Mixed mode simulation also greatly increases the capacity of the machine. Simulating function blocks at a high level leaves more of the capacity of the machine available for the rest of the circuit. The YSE's *array processors* simulate memories at a high level. The logic simulation machine proposed by Abromovici et.al. [20,21] incorporates a more complete mixed mode simulation facility.

2.5 Weaknesses of Conventional Simulation Engines

Existing simulation machines suffer from two major weaknesses: 1) they do not accurately model MOS circuits and 2) they bind circuit elements to simulation hardware at compile time.

Logic simulators are inadequate for many MOS circuits which cannot be described in terms of gates. Designers often attempt to force these circuits into a logic model by using several gates to model one transistor. [22] Some logic simulators attempt to provide features such as strengths which mimic the behavior of a switch level simulator. These approaches in general do not model the subtleties of MOS circuits. Switch-level simulation is fundamentally different from logic simulation. Switch level simulation operates by tracing paths to solve for the steady state response of a circuit. This steady state response is difficult to compute using logic gate models.

All existing simulation machines bind circuit elements to simulation hardware at compile time. In the YSE, even the interconnection performed by the crossbar switch is determined at compile time. This compile time binding limits the size of circuits which can be simulated to the size of the available hardware. Compile time binding also limits the effective concurrency by forcing processors bound to idle subcircuits to remain idle.

The design of the MSE incorporates the speedup techniques used by logic simulation engines and uses the MOSSIM algorithm and run-time binding to overcome their weaknesses. Run time binding attempts to solve the problem of load balancing. With run-time binding of circuit subnetworks to hardware, a virtual processor mechanism can be implemented where each subnetwork is assigned to a virtual processor, but only the active virtual processors require physical hardware. The use of virtual processors also eliminates an upper bound on the size of a circuit which can be simulated. The virtual processor mechanism is described in more detail in section 3.

3 Algorithm

The MSE implements the switch-level algorithm described previously [11], referred to here as the "MOSSIM" algorithm. Some modifications were made to the data structures and control flow used in the

simulator MOSSIM II, a software implementation of the algorithm, to better exploit the characteristics of hardware. This algorithm, is based on a formal switch-level model of MOS transistor networks. By modeling MOS transistor ratios and node capacitances and by considering an MOS transistor as a truly bidirectional device, MOSSIM provides considerably more accurate simulation of MOS LSI than conventional logic simulators. MOSSIM achieves performance comparable with logic gate simulators by using an event-driven scheduling algorithm that exploits the locality of events in a circuit. We will give only an overview of the switch-level model and theory in this paper.

3.1 Model

A switch-level network consists of a set of nodes connected by transistors. An *input* node provides a strong, externally-set signal much like a voltage source in an electrical circuit. All other nodes are *storage* nodes, able to store a value in the absence of an applied input and to share charge with other storage nodes. Nodal capacitances are modeled by assigning each storage node a *size*, from the set $\{\kappa_1, \ldots, \kappa_{max}\}$ according to the relative capacitance of the node compared to other nodes in the network. Node sizes are ordered, $\kappa_1 < \cdots < \kappa_{max}$. When a set of storage nodes share charge, the resulting state is determined by the state(s) of the strongest node(s). Input nodes are indicated by a size ω. Node voltages are represented by states 0 (low), 1 (high), and X (invalid or uninitialized.)

A transistor is a device with terminals labeled, "gate", "source", and "drain" that acts as a resistive switch connecting the source and drain nodes controlled by the state of the gate node. All transistors are viewed as bidirectional elements with no predetermined direction of information or current flow. A transistor has a *type* indicating the conditions under which it will become conducting and a *strength* indicating its conductance relative to other transistors in a ratioed circuit. Transistor types n and d conduct when the gate node is in state 1, while transistor types p and d conduct when the gate node is in state 0. When the gate node of an n or p transistor is in state X, the transistor is modeled as having arbitrary conductance between fully conducting and open circuited. Transistor conductances are modeled by assigning each transistor a strength from the set $\{\gamma_1, \ldots, \gamma_{max}\}$ where strengths are ordered $\gamma_1 < \cdots < \gamma_{max}$. In a ratioed circuit consisting of paths of conducting transistors from several input nodes to a storage node, the state of the node is determined by the state(s) of the input node(s) connected by maximum strength paths, where the strength of a path equals the minimum transistor strength in the path. Transistor conduction conditions are represented by states 0 (nonconducting), 1 (fully conducting), and X (between nonconducting and conducting).

3.2 Data Structure:

A switch-level network is represented by three data structures in the MSE, in a manner similar to an adjacency list representation of a graph. These data structures are shown in figure 2. In fact the simulation algorithm treats the network as a *dynamic multigraph*, where each transistor, depending on its state, can create an edge from its source to its drain and from its drain to its source. A fourth data structure, the stack memory STKM is used to schedule nodes for evaluation. The node list memory, NLM, contains a record for each node, with fields indicating the parameters of the node (size, type, and state), a pointer to the fanout list (representing the set all transistors for which this node is the gate), and a pointer to the link list (representing the set of all transistors for which this node is the source or drain.) The link list memory, LLM, contains a pair of link records for each transistor: one providing a pointer from the source node to the drain, and one providing a pointer from the drain node to the source. Each link list record also contains fields indicating the transistor parameters (type, strength, and state.) The records in this memory are organized in *link lists*, blocks of contiguous memory locations

each containing the set of link records pointing from a given node to all adjacent nodes. The gate list memory GLM contains *gate records*, pointers to the links representing transistors for which this node is the gate. The gate records are organised in *fanout lists*, blocks of contiguous memory locations each containing the set of gate records pointing to all links representing transistors for which a given node is the gate. Thus, every transistor in the network requires two link records and two gate records in the data structure. By describing the connectivities in the network as sets of pointers organised in blocks, the hardware can traverse the network by following pointers and enumerate connections by sequencing through memory with a counter. By comparison, MOSSIM II also represents the network connectivities as sets of pointers but organises them in linked lists. We chose a different representation for the MSE to reduce the overhead of the extra memory required to store pointers in a linked list.

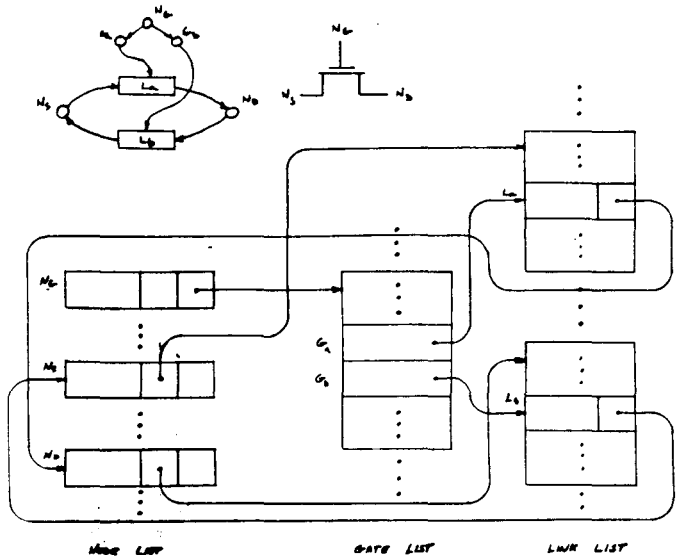


Figure 2. MSE Network Data Structure

3.3 Algorithm

To simulate a circuit over a sequence of input vectors, the MSE proceeds as follows:

```
Set all nodes to X
For each input vector do
begin
  Update input nodes
  do begin
    1. Logic Update
    2. Perturbation
    3. Blocking Strength
    4. Up/Down Strength
  end
  until stable state reached or step limit exceeded
end
```

MSE ALGORITHM - MAJOR LOOP

To simulate the response of a circuit to a new set of input values, the MSE repeatedly computes the

new excitation states and sets the nodes to their excitation states until a stable state is reached (i.e. the current states equal the excitation states), or a user-specified step limit is exceeded. Computing the excitations involves setting the transistors according to the states of their gate nodes (step 1 in the above program), and computing the *steady state response* of the nodes (steps 2-4), i.e. the new states formed on the storage nodes due to the connections from input nodes and other storage nodes formed by the conducting transistors.

In general, the algorithm need only compute the steady state response of those nodes that could be affected by the changing conduction states of the transistors and the changing states of the input nodes. All other nodes will have steady state responses equal to their current states. By recomputing the node states only in the active portions of the network, a switch-level simulator can achieve a performance comparable to event-driven logic gate simulators.

Unlike logic simulators that propagate logic values through a gate network, MOSSIM computes the steady state response by analysing the paths in the graph with edges for each transistor in the 1 or X state between its source and drain nodes. The steady state responses of the nodes are determined by the paths in the graph as follows. A *path* consists of a root node and a (possibly empty) sequence of edges to a destination node. The strength of a path is defined as the minimum of the size of the root and the strengths of the transistors corresponding to the edges, where strength values are ordered $\kappa_1 < \cdots < \kappa_{max} < \gamma_1 < \cdots < \gamma_{max} < \omega$. This ordering of node sizes relative to transistor strengths reflects the fact that a path of conducting transistors from an input node to a storage node can override any stored charge, and that the state of an input node is not affected by the network connections. A path is *definite* if none of its edges correspond to transistors in the X state. A path is *blocked*, if for some initial segment of the path (i.e. a path consisting of the same root node and a subset of the edges) and for some definite path with the same destination as the initial segment, the strength of the definite path is greater than that of the initial segment. For a given storage node, its steady state response is determined by the current states of the nodes at the roots of the unblocked paths having this node as destination. That is, the steady state response equals 0 (respectively 1) if the current states of all the root nodes equal 0 (respectively 1), and equals X otherwise. It has been shown that this definition in terms of unblocked paths handles ratioed circuits, dynamic charge storage, charge sharing, attenuating pass transistors and unknown logic states in a very accurate manner [11].

The four steps in the above program perform the following functions. During the *Logic Update* step, the conduction states of the transistors whose gate nodes have changed state are updated, and the source and drain nodes of these transistors are queued for the next step. During the *Perturbation* step, the set of all nodes that could be affected by the changing transistor states are found by starting at the nodes queued in the logic update step and traversing the links representing transistors in the 1 or X state. Any path containing an input node other than at the root must be blocked. Hence, the links leading out from an input node are not traversed. Each time a new node is encountered, it is added to the queue. This process continues until no further nodes are found.

The two remaining steps serve to compute the steady state response for all nodes in the queue. The *Blocking Strength* step determines the strength of the strongest definite path to each node. This computation proceeds in a manner similar to Dijkstra's shortest path algorithm [23], except that rather than finding the path that minimizes the sum of the edge lengths, it finds the path that maximizes the minimum edge strength. That is, for every node n, the value $block(n)$ is initialized to the size of the node and all nodes are placed in a set S. The computation proceeds iteratively by selecting and removing a node n from S that maximizes $block(n)$ and for each link l in the link list for n with state 1 pointing to a node m in S, it computes $block(m) \leftarrow \min(block(n), strength(l))$]. This process continues until the set S is empty. The *Up/Down Strength* step computes the up (respectively down) value for each node, i.e. the strength of the strongest unblocked path to the node having a root node in the 1 or X (respectively 0 or X). If no such path exists, the value is set to 0, where $0 < \kappa_1$. This computation also proceeds in a manner similar to Dijkstra's algorithm.

For each node n, the value $up(n)$ (respectively $down(n)$) is initialised to the size of n if the current state of node n is 1 or X (respectively 0 or X) and the size of n is greater than or equal to $block(n)$. Otherwise $up(n)$ (respectively $down(n)$) is initialised to 0. Starting with the set S containing all nodes, a node n is selected that maximises the value of $\max(up(n), down(n))$ and removed from S. For each link l in the link list for n with state 1 or X pointing to a node m, it computes

$$up(m) \leftarrow \begin{cases} \max[up(m), \min(up(n), strength(l))], & \min(up(n), strength(l))] \geq block(m) \\ 0, else \end{cases}$$

and similarly for computing $down(m)$. If this computation causes the value of $up(m)$ or $down(m)$ to change, m must be added back to S if it had previously been removed.* Once this computation terminates, the steady state response of node n equals 1 if $down(n)$ equals 0, 0 if $up(n)$ equals 0, and X otherwise.

To map the MOSSIM algorithm into hardware, each of the four steps described above was manipulated to fit into a single algorithm template, given below. This factoring of code across the simulation steps allows the same control logic to be used for each simulation step, a criterion more important in hardware design than in software. During the 4 steps, different operations of operations are performed, different types of records are followed, and different scheduling criteria are used (indicated by the statements in square brackets), but the overall flow of control is identical.

```
For each scheduled node n:
  Operate on n
  [schedule n for next step]
  For each active record r in some list for n:
    Operate on dest(r)
    [schedule dest(r) for this step]
    [schedule dest(r) for next step]
```

MSE - ALGORITHM TEMPLATE

During the logic update step, no operation is performed on n, but n is scheduled for the next step. Then for each record in the fanout list for n, the state of the link pointed to is updated. During the perturbation step, the values of $block(n)$, $up(n)$, and $down(n)$ are initialized, and n is scheduled for the next step. Then for each record in the link list for n having state 1 or X, the node pointed to is scheduled for the current step. During the blocking strength step, nodes are removed from the queue in decreasing order of their *block* values, and the node is scheduled for the next step. Then for each record in the link list having state 1, the *block* value of the node pointed to is updated as described before. During the up/down strength step, nodes are removed from the queue in decreasing order of the maximum of their *up* and *down* values. For each record in the link list having state 1 or X, the *up* and *down* values of the node pointed to are updated as described earlier. If either of these values changes, the node is scheduled for the current step. The new state of the node is computed (even though it may need to be recomputed), and if different from the current state, the node is scheduled for the next step (the up/down step of the next iteration.)

As can be seen by the preceding discussion, the MSE algorithm differs in many respects from conventional logic gate simulators.

---

*These changes to the control flow of Dijkstra's algorithm are required to compute both the up and down values simultaneously.

The fundamental operations involve traversing graphs and computing path strengths. The order in which nodes and links are traversed depends on the dynamic settings of the transistor states, and hence an architecture with a predetermined control flow such as the YSE would not be appropriate. Like logic gate simulation, however, the fundamental operations of switch-level are quite simple, and the algorithm models all of the nuances of MOS circuits with a single computational paradigm (in terms of unblocked paths). These properties permit a cost-effective hardware implementation that improves greatly on the performance of software switch-level simulators.

3.4 Virtual Processors

The very locality of activity that enables subcircuit concurrency and localized steady state response computation can lead to a degradation in performance due to idle processors. If the amount of activity in each processor is not equal, processors with low activity will complete a processing step early and remain idle until all processors complete the step. To avoid this potential degradation, we have developed the concept of *virtual processors (VPs)*. This concept is analogous to that of *virtual memory* [24]. We partition the circuit into many more subcircuits than we have physical processors. A virtual processor, associated with each subcircuit, contains the complete state of the simulation of that circuit. To maximize throughput, the VPs are dynamically mapped to *physical processors (PPs)* based on activity. As soon as a VP becomes idle, it is *swapped out* to be replaced by a VP with activity. This mechanism minimizes the amount of time a PP is idle. This virtual processing method also permits a trade-off between the amount of hardware (i.e. number of physical processors), and the speed of simulation, without affecting the maximum size circuit that can be simulated. In order to implement the virtual processor concept, a swapping mechanism, mapping mechanism, and scheduling algorithm are required.

Swapping is implemented at the VP level by copying the entire state of a VP into backing store. If we restrict swapping to occur at the completion of the outer loop of each algorithm, none of the PP's working registers need be saved or restored. Only the node list, link list, gate list and event stacks (including stack pointers) need be copied. After the old VP is swapped out, the new VP is swapped in by copying its state from backing store.

Mapping is required on all interprocessor communications. An individual VP uses only local addresses and requires no mapping. In fact, as long as it is not moved to a different processor group, a VP need not know which PP it is executing on. Mapping is implemented in the interprocessor message switch. All messages are transmitted with virtual addresses. The message switch queues arriving messages according to virtual address. A separate message queue is maintained for each VP. A routing table in the message switch holds the current virtual processor to physical processor mapping and is used to direct output from the queues.

An additional benefit of using short local addresses within VPs and longer global addresses only for interprocessor communication is that the storage requirements for pointers is greatly reduced. Also, the hierarchical addressing mechanism of the MSE is implemented so that it need not be limited to two levels of hierarchy. Messages could be passed between message switches using a third level of addressing. Thus, the size of network the MSE can simulate is not limited by address size.

A Scheduling Algorithm controls the assignment of virtual processors to physical processors. An efficient scheduling algorithm should optimize throughput by keeping all physical processors busy all the time. A candidate scheduling algorithm is shown below:

```
Starting From an Initial Assignment
While some virtual processor is not done
```

```
If a process, p1, on processor P is done
  Select the swapped out process, p2, with the
  longest queue
  Swap p1 out of processor P
  Swap p2 in to processor P
```

VIRTUAL PROCESSOR SCHEDULING ALGORITHM

4 Architecture

The MSE achieves its performance through subnetwork concurrency, functional concurrency and specialization. Subnetwork concurrency involves partitioning the network into several subnetworks and simulating the subnetworks in parallel. Within each processor functional concurrency is achieved by performing the operations of scheduling, node evaluation and network traversal in parallel. Finally, in performing each of these operations specialized logic circuits are used to implement the time critical functions offering orders of magnitude speedup over general purpose computer instructions.

As shown in figure 3, the MSE consists of a number of subnetwork processors (SP) connected by a message bus (MB) to a message switch (MS). Auxiliary processors (AP) may also be connected to the MB to perform functional simulation. A host processor (HP) is connected to all processors by the host bus (HB). The HP controls the operation of the MSE, performs virtual processor swapping and has the ability to read and write each register and memory location in the machine.

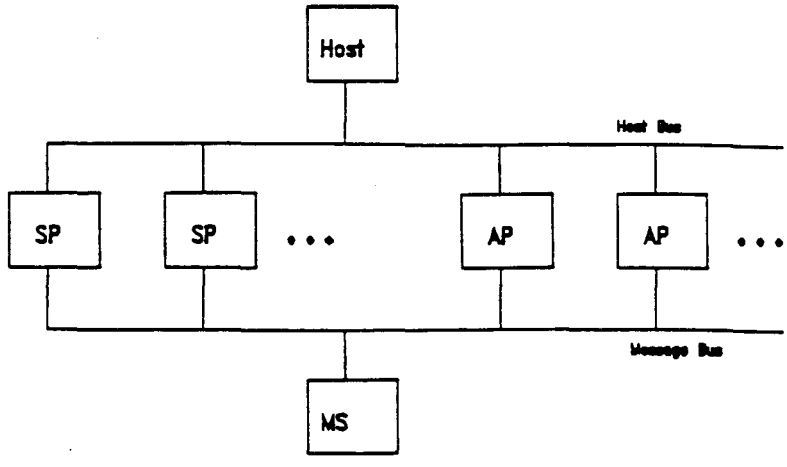


Figure 3. MSE Block Diagram

The SPs and APs simulate subnetworks of a circuit in parallel. Interactions between subnetworks create messages which are routed through the MS. The MS performs a virtual subnetwork to physical SP translation for each message and queues messages for subnetworks which are swapped out. Simulation studies indicate that up to eight SPs may be attached to a single MB/MS without significant degradation due to bus contention.

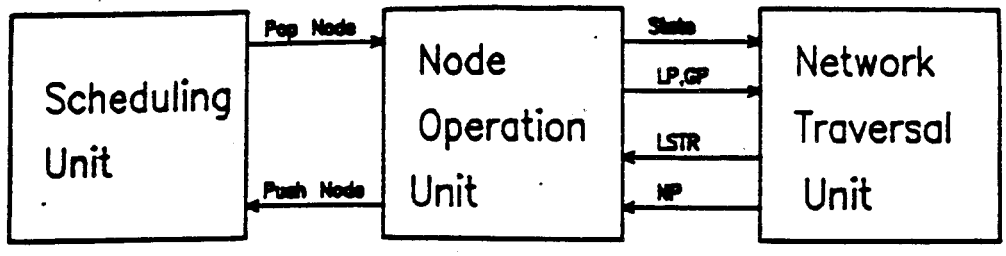


Figure 4. Subnetwork Processor Block Diagram

The SP, shown in figure 4, is the hardware kernel of the MSE. Each SP has a capacity of 4096 nodes and 16384 transistors partitioned into separate physical processors of 1024 nodes each. An SP implements the MOSSIM algorithm performing all operations for its subnetwork and sending messages to the MS for operations involving other subnetworks. To exploit all possible functional concurrency a separate function unit is associated with each major data structure. The scheduling unit (SU) implements the scheduling priority queues. The node memory and a path strength unit which operates on nodes are contained in the node operation unit (NOU). The network traversal unit (NTU), contains the link and gate lists which describe the transistors and the network connectivity. The SP also contains two additional function units. The control processor supervises operation of the SP, and the input/output unit handles inter-processor communication. Specialized hardware was added to each unit as necessary to balance their performance so that no one unit was a bottleneck.

Node Operation Unit (NOU)

The NOU manages the node list data structure performing operating on pairs of nodes as directed by the SU and NTU. The NOU, shown in figure 5, consists of a node list memory (NLM), associated addressing registers, and a node data path containing a path strength unit (PSU), and source (SNR), destination (DNR) and result (RRR) registers.

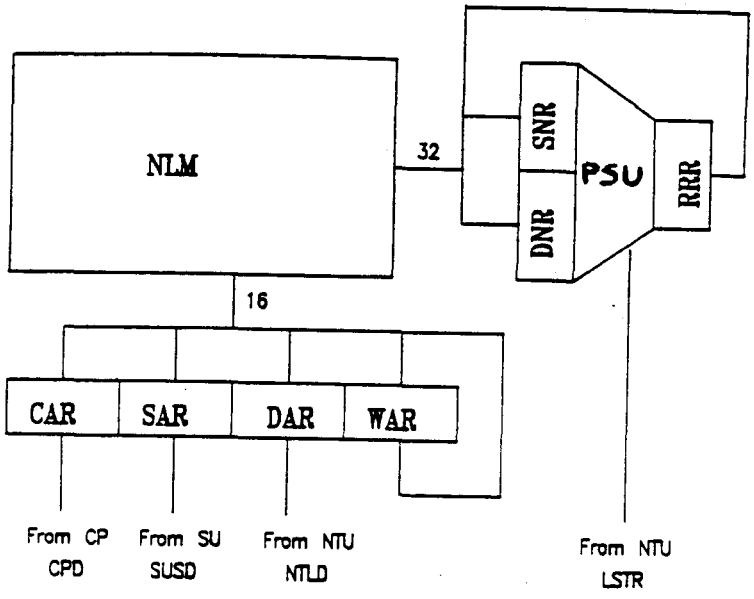


Figure 5. Node Operation Unit Block Diagram

This hardware performs the node operations used to trace paths in the network. It operates by selecting the highest priority node from the active list and computing the effects of this *source node* on all connected *destination* nodes. If a destination node is updated as the result of this operation, it is added to the active list. First the NOU requests the highest priority node from the SU. The SU responds by returning the address of this *source* node, N_S. The address is latched in an addressing register and used to read the source node record from the NLM. This record is latched in the SNR and its link and gate pointers are passed to the NTU. The NTU returns the addresses of nodes which are connected to N_S through an on or unknown transistor. As each of these *destination* node addresses is returned to the NOU, it is latched in an addressing register and used to read a destination node, N_{Di}, from the NLM. After the node record for N_{Di} is latched in the DNR, the PSU performs the path strength computation and the result is latched in the RRR and then written back to the NLM. If the destination node is updated, its address is placed on the active list so the change can be propagated to its neighbors.

Operations on destination nodes are pipelined four ways as shown in figure 6 below. While the address of the fourth destination node N_{D4} arrives from the NTU, the record for N_{D3} is read from the NLM, nodes N_S and N_{D2} are operated on in the PSU, the results from operating on destination node N_{D1} are written back to the NLM. Pipelining allows a path strength operation to be performed once per clock cycle after the pipeline is full. There is some overhead, however, for loading new source nodes. This pipelining is an example of fine-grain functional concurrency (generate address, fetch, operate, write result) nested within the higher level functional concurrency of the SU, NOU and NTU.

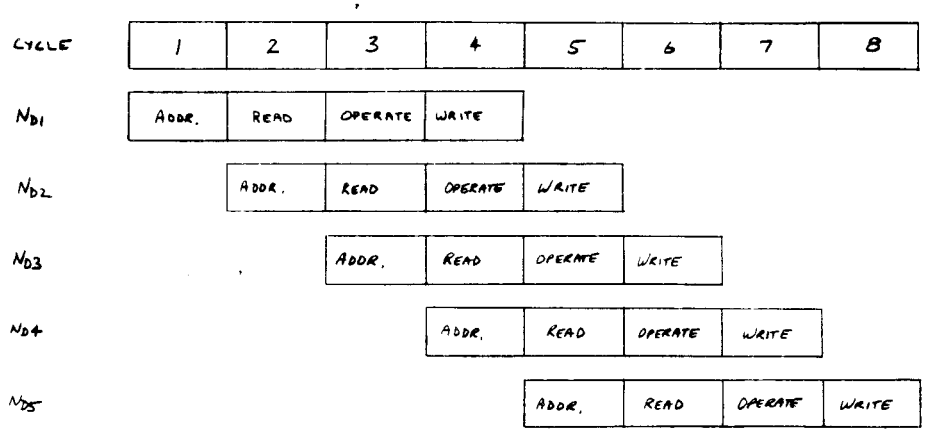


Figure 6. Destination Node Operation Pipelining

Specialization is also used to accelerate the operation of the NOU. The PSU hardware performs the path strength operation in 100ns. This operations requires $\approx$ 30 lines of Mainsail [25] code, or $\approx 150\mu s$ to perform in software. This speedup, by a factor of 1000, appears to greatly outweigh the contribution of pipelining; however, without pipelining it is not possible to get data to the PSU fast enough to take advantage of its performance. Both functional concurrency and specialization are required to achieve balanced performance.

Scheduling Unit (SU)

The SU manages the priority queue data structure used for scheduling operations on nodes. The priority queues are implemented using buckets. Each bucket is a linked list stack allocated from a free list. A separate stack is used for each priority level for both the current and next simulation steps. Since nodes are scheduled according to signal strength, there are sixteen stacks for each step, one stack for each strength. Hooks are provided to add timing simulation by allocating a separate stack for each time increment in the logic update step.

As shown in figure 7, the SU consists of a stack memory (STKM), a stack pointer memory (SPM), and a free pointer (FP). To support a linked list structure, each entry in the STKM contains two fields, data and next. The FP points to the beginning of the free list and each SP in the SPM points to to top of a scheduling stack. A strength counter (SC) keeps track of the strongest node scheduled for the current step so that nodes can be retrieved in order of strength. An empty comparator (EC) checks pointers for a null value.

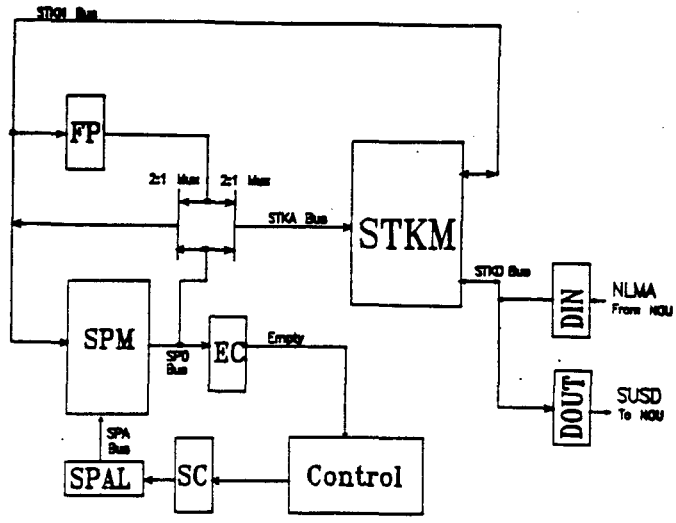


Figure 7. Scheduling Unit Block Diagram

SU operations are initiated by NOU requests to pop the highest priority node from the current step's priority queue or to push a node on either the current or next step's queue. The push operation takes place in one clock cycle. During the read half of the cycle, the FP is used as an address to read the head of the free list from the STKM. At the same time, the strength of the signal being pushed and the current simulation step are used as an address to read the appropriate stack pointer (SP) from the SPM. During the write half of the clock cycle, the links are adjusted ($SP \leftarrow FP$, $FP \leftarrow NEXT$, $NEXT \leftarrow SP$). Then the data and the updated links are written.

The pop operation proceeds in two steps: finding the highest priority non-empty stack, and then removing the head of this stack. The SC, an estimate of the highest priority for which a node is scheduled, is used to address the SPM and is decremented until a non-empty stack is found. Since the SC tracks each push, it is always an upper bound on the highest non-empty priority. Once a non-empty stack is found, the data is read and the pointers are adjusted by reversing the assignments listed above for push.

Specialization is used in the SU to match its speed to the requirements of the NOU. A push takes only one clock cycle, and since the initial value of the SC almost never finds an empty stack the pop operation typically takes two clock cycles. Implementing these operations in software would take about 10 assembly language instructions or 10μs.

Network Traversal Unit (NTU)

The NTU manages the network topology data structure. It traverses the adjacency list of the current source node returning pointers to destination nodes. During the logic update step the NTU updates the state of transistors controlled by nodes which have changed states. The two operations correspond to the link list and gate list data structures shown in figure 2.

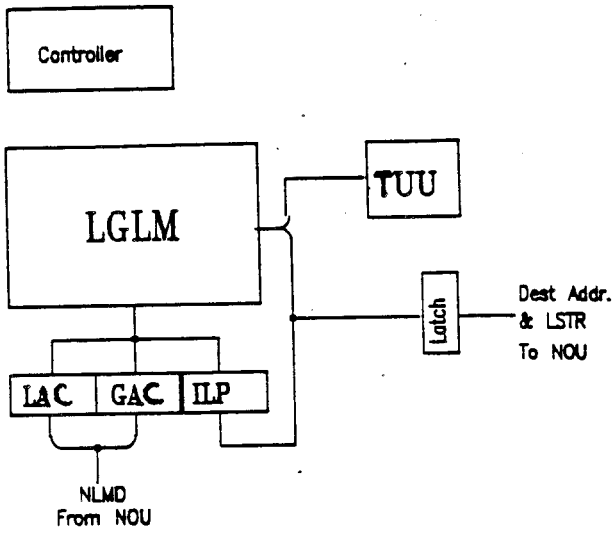


Figure 8. Network Traversal Unit Block Diagram

A block diagram of the NTU is shown in figure 8. The link and gate lists for each node are stored as arrays in the link and gate list memory (LGLM). The LGLM is addressed by three registers: the gate address counter (GAC), the link address counter (LAC) and the indirect link pointer (ILP). A transistor update unit (TUU) updates transistor states.

In operation, the NOU passes the link and gate pointers of the source node, N_S, to the NTU. These pointers are latched into the LAC and GAC. To return an adjacency list the LAC is incremented each cycle to sequence through the source node's link list. Each link in the list is examined to determine if it is active and if it is local or external. The pointers from local active links are returned to the NOU. External active links initiate a message transmission to the destination node's virtual processor.

During the LU step, the NTU sequences through a node's gate list and updates all the destination link records with their new state. First the source node's gate pointer is latched in the GAC. As the GAC sequences through the gate list, the link pointer in each gate record is latched in the ILP. This link is then read from the LGLM, modified in the TUU and written back. If the state of a link changes, a pointer to it's destination node is passed back to the NOU so it can be scheduled for reevaluation.

Arrays are used to implement the lists in the NTU because these lists are static. Linked lists are used in the SU to allow memory to be dynamically allocated among the lists. Implementing a list as an array allows it to be quickly sequenced by a counter and avoids the overhead of storing next pointers; however changing the size of an array requires copying. While the linked list implementation incurs the overhead of next pointers it is required in the SU where the distribution of memory among the event lists varies considerably with time.

4.2 Message Switch

The MS routes and queues messages caused by external link and gate records in the NTU. It performs a virtual subnetwork to physical SP translation for each message and queues messages for subnetworks which are swapped out.

As shown in figure 9, The message switch consists of an input FIFO, a mapping memory (MM), a queue memory (QM), an output port and a message control processor (MCP). Messages from SPs arrive and are queued in the input FIFO. Each message in turn is then processed by looking up the location of its destination VP in the MM. If the destination VP is resident in a PP, the PP address replaces the VP address in the message, and the message is routed to the output port. If the destination processor is swapped out, the message is queued in the QM. When a processor is swapped in, the MCP transmits its queued messages over the output port.

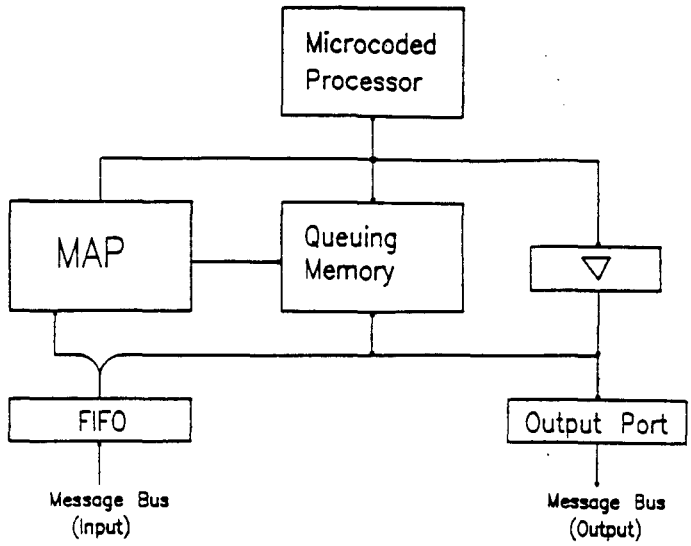


Figure 9. Message Switch, Block Diagram

4.3 Auxiliary Processor

An auxiliary processor (AP) is planned to provide special purpose hardware to improve the speed of event driven functional simulation. Simulation studies using the DSIM functional simulator [26] have suggested a number of areas where special purpose hardware can accelerate functional simulation. A scheduling unit and network traversal unit similar to the units in the SP would speed up event list management and fanout list traversal. Hardware can speed up message passing between virtual processors and interfacing functional signals to switch-level signals. A signal manipulation unit would accelerate extracting subfields of signals and resolving conflicts on signal nodes. Also models for commonly used functional units such as ROMs RAMs and PLAs could be accelerated by hardware.

A block diagram of the AP is shown in figure 10. An input/output unit (IOU), handles communication with the message switch. An scheduling unit, (SU), performs event scheduling for logic updates. When an event occurs, the SU sends a signal pointer to the NTU which scans the sensitization list for each signal to determine which functional blocks must be resimulated. Simple functional blocks such as ROMs, RAMS and PLAs will be simulated by an array evaluation unit (AEU). More complex function blocks will be simulated by a 68000 microprocessor.

Adding a functional simulation capability to a simulation accelerator is important for two reasons. First, it allows large circuits to be simulated quickly using mixed mode simulation. More importantly, it allows the input vectors to be generated and output vectors to be checked by functional models within the machine. Without this capability setting input vectors and checking output vectors in the host processor can become a bottleneck.

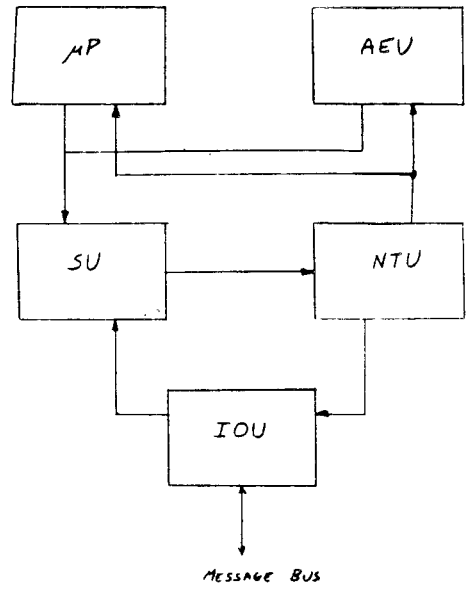


Figure 10. **Auxiliary Processor, Block Diagram**

4.4 Host Processor

The host processor (HP) acts as the user interface to the MSE and the global MSE controller. The host processor performs the following functions:

- command interpretation,
- setting nodes for input vectors and watching nodes for output vectors,
- loading and partitioning the circuit model,
- coordinating SP, MS, and AP operation,
- debugging and system diagnostics.

Currently the HP is a SUN-1 workstation running the V-Kernel. We are planning on replacing this host in the near future with a SUN-2 workstation running UNIX.

5 Performance

The performance of the MSE has been measured at 1.5M path strength operations per second (PSOPS). This corresponds to about 150K GEPS for gate oriented circuits, about 300 times faster than a VAX 11/780 running MOSSIM II. [12]

Compared to the other simulation engines shown in table 1, the MSE appears to be an order of magnitude lower in performance. However, efforts to accurately simulate MOS circuits using conventional logic simulators [19, 22] generally require four to six gates to model a single transistor, and several passes through these gates to model a transistor state change. Furthermore, such models are usually not capable

of modeling many MOS circuit phenomena such as charge sharing, and sneak paths. Thus, for accurate MOS simulation, the MSE offers better performance than any existing simulation machine.

The performance figure of 1.5M PSOPS was measured on the prototype MSE hardware using small benchmark circuits. While the MSE can achieve speeds of 5M ROPS when its node operation pipeline remains full and no input nodes are encountered, in the benchmark circuits these ideal conditions rarely occurred. In practice the each node has an average of three neighbors. As soon as the pipeline is filled for one source node, it must idle for three cycles while the next source node is loaded. Also, typically one quarter to one third of the nodes encountered during simulation are input nodes. When an input node is a destination node, the node operation is performed in the opposite direction requiring an additional cycle.

The relationship of 10 path strength operations per logic event was determined by functional simulation of larger benchmark circuits containing a few thousand transistors. The MSE functional simulator is a 5000 line Mainsail program which simulates the MSE at the register transfer level. [25] Across a wide range of circuits ranging from 20 nodes to 2000 nodes in size, functional simulations show that the MSE requires an average of 10 path strength operations per logic event. A logic event is defined as a logic update of a transistor gate. This type of event often triggers re-evaluation of an entire logic gate and corresponds well with evaluating a multi-input gate.

A breakdown of where the MSE spends its time is shown in figure 11. Only 10%of the time is spent in the logic update step while over 60%of the total time is spent in the path tracing steps. The design of the MSE has been influenced by these statistics with an emphasis being placed on improving the performance of the path finding process. In the future, we believe that the most performance improvement will result from optimisations in the perturbation step which will reduce the number of nodes which must be re-evaluated during the relaxation steps.

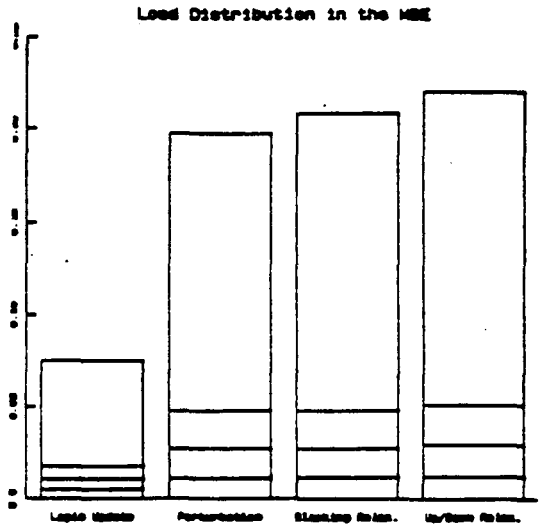


Figure 11. MSE Activity Profile

While we have run some simulations of MSEs with several SPs, we have not taken sufficient data to make any conclusions about the efficiency of multiprocessing or the performance of the virtual processor load balancing mechanism.

6 History and Status

The MSE project was begun in July 1983 with a study of static and dynamic locality in MOS transistor networks. Based on this study, the architecture was defined in early August. To test the architecture the MSE functional simulator was written during August and was operational by early September. From September to February 1984, the an implementation of the MSE SP was designed using standard catalog parts. The design uses 396 integrated circuits packaged on a single wire-wrap board. The board was wired in May and debugging was completed in October of 1984.

Much work remains to be done. A great deal of systems software must be written before the MSE can be used by designers to simulate their chips. It would be valuable to construct a multiprocessor MSE with additional SPs and a MS to test the idea of virtual network processing. Experiments must be performed to determine message and swapping traffic and their dependency on the ratio of physical SPs to virtual subnetworks. More work is also needed on building processors to accelerate functional and circuit simulation so the MSE can be incorporated in a mixed-mode simulation environment.

7 Conclusion

We have designed and constructed the MOSSIM Simulation Engine, a special purpose processor to accelerate switch-level simulation of MOS VLSI circuits. The MSE overcomes two limitations of existing simulation engines: The MSE, by using switch level models, provides greater accuracy when simulating MOS circuits. MOS effects such as charge sharing, sneak paths and dynamic storage are correctly simulated. By using virtual network processing the MSE is able to simulate circuits larger than the size of the simulation machine. We conjecture that virtual network processing makes more efficient use of parallel processors.

For a problem to be a candidate for special purpose hardware, it must be computationally demanding, have a stable and structured algorithm, have potential parallelism (both functional and structural), and have some operations which are poorly matched to the capabilities of a general purpose computer. Speedup is achieved through specialization and concurrency. This speedup must be balanced so that specialized logic does not idle waiting for data.

The implementation of special purpose hardware should not simply follow a software implementation. The cost/performance characteristics of hardware and software are very different. In hardware it is important to fit the problem into a uniform execution mechanism and then to accelerate this uniform mechanism. For the MSE, the MOSSIM algorithm was modified to fit an algorithm template. The hardware of the machine was then designed to optimise execution of the template.

The ideas incorporated in the MSE can be applied to many problems of a similar nature. Many algorithms which appear to be irregular can in fact be fit to a template which can form the basis for special purpose hardware. The concept of virtual network processing, run-time binding of sub-networks to hardware, can be applied to any problem which is characterized by sparse clustered activity. Other problems which are candidates for special purpose hardware and virtual network processing include circuit simulation, geometry compaction, and routing. It is interesting to note that these problems also use a sparse dynamic graph data structure and could make use of the scheduling and network traversal units of the MSE.

One direction of future research is to construct more flexible hardware accelerators. There are far too many demanding applications to construct special purpose hardware for each application. We propose constructing special purpose hardware to accelerate operations on common data structures. By combining these accelerators in different ways designers can share hardware much in the same way that programmers share code. For example, many of the components of the MSE are involved in graph

operations such as path finding. These components could be packaged as a graph accelerator and combined with other accelerators to address a wider range of problems.

8 Acknowledgements

We thank Chuck Seitz for providing support and guidance to this project and Hsiu-Tung Yu for contributing to the software and hardware debugging. This research was sponsored by the Defense Advanced Research Projects Agency, ARPA order number 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597.

References:

[1] Dally, W.J., *The MOSSIM Simulation Engine: Architecture and Design*, Caltech Technical Report 5123:TR:84, April 1984.

[2] Dally, W.J. and R. Bryant, "A Special Purpose Processor for Switch-Level Simulation," *IEEE International Conference on Computer-Aided Design*, 1984, pp. 242-244.

[3] Pfister, G. F., "The Yorktown Simulation Engine: Introduction," *19th Design Automation Conference*, ACM, 1982, pp. 51-54.

[4] Denneau, M. M., "The Yorktown Simulation Engine," *19th Design Automation Conference*, ACM, 1982, pp. 51-54.

[5] Kronstadt E. and G. Pfister, "Software Support for the Yorktown Simulation Engine," *19th Design Automation Conference*, ACM, 1982, pp. 51-54.

[6] "ZyCad LE-001 and LE-002 Product Description," ZYCAD,1982.

[7] Bartow, R. L. *et al.*, "Architecture of a Hardware Simulator," *IEEE Conference of Circuits and Computers*,1980, pp.891-893.

[8] "Daisy Megalogician, Product Description," Daisy Systems, 1984.

[9] Bryant, R., *A Switch-Level Simulation Model For Integrated Logic Circuits*, Ph.D. dissertation, MIT, 1981.

[10] Bryant, R., "MOSSIM: A Switch-Level Simulator For MOS LSI," *18th Design Automation Conference*, ACM, 1981, pp. 786-790.

[11] Bryant, R., "A Switch-Level Model and Simulator for MOS Digital Systems," *IEEE Transactions on Computers*, vol. C-33, pp. 160-177, February 1984.

[12] Bryant, R., M. Schuster and D. Whiting, *MOSSIM II: A Switch-Level Simulator for MOS LSI, User's Manual*, Caltech Technical Report 5033:TR:82, January 1983.

[13] Szygenda, "TEGAS-2 - Anatomy of a General Purpose Test Generation and Simulation System for Digital Logic," *9th ACM-IEEE Design Automation Workshop*, 1972, pp. 116-127.

[14] Vladimirescu, A. *et al.*, *SPICE Version 2G.5 User's Manual*, University of California, Berkeley, Technical Memo.,August 1981.

[15] Chawla, B., H. K. Gummel, and P. Kozah, "MOTIS - an MOS Timing Simulator," *IEEE Transactions on Circuits and Systems*, vol. CAS-22, December 1975, pp. 901-910.

[16] Newton, A. R., "Techniques for the Simulation of Large-Scale Integrated Circuits," *IEEE Transactions on Circuits and Systems*, vol. CAS-26, September 1979, pp. 741-749.

[17] Ruehli, A. E., and G. S. Ditlow, "Circuit Analysis, Logic Simulation, and Design Verification for VLSI," *Proceedings of the IEEE*, Vol. 71, No. 1, January 1983, pp. 34-48.

[18] Breuer, M. A. and A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems.*, Computer Science Press, 1976.

[19] Barzilai, Z. *et al.*, "Simulating Pass Transistor Circuits Using Logic Simulation Machines," *20th Design Automation Conference*, ACM, 1983, pp. 157-163.

[20] Abraomovici, M. et al.,"A Logic Simulation Machine," *19th Design Automation Conference*, ACM, 1982, pp. 65-73.

[21] Abramovici, M. et al.,"A Logic Simulation Machine," *IEEE Transactions of Computer-Aided Design of Integrated Circuits and Systems*, Vol CAD-2, No.2, April 1983, pp. 82-94.

[22] Sherwood, W., "A MOS Modelling Technique for 4-State True-Value Hierarchical Logic Simulation," *18th Design Automation Conference*, ACM, 1981, pp. 775-785.

[23] Aho, A.V., J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[24] Denning, P., "The Working Set Model for Program Behavior," *Communications of the ACM, Vol. 11, No. 5*, May 1968, pp.323-333.

[25] Wilcox, C. R., M. L. Dageforde, and G. A. Jirak, *Mainsail (TM) Language Manual Version 4.0*, XIDAK, 1979.

[26] Dally W.J., *The DSIM Functional Simulation System: Preliminary User's Manual*, Caltech Display File 5109:DF:83, December 1983.